

Java et Eclipse

Développez
une application Java

Henri LAUGIÉ

Fichiers à télécharger



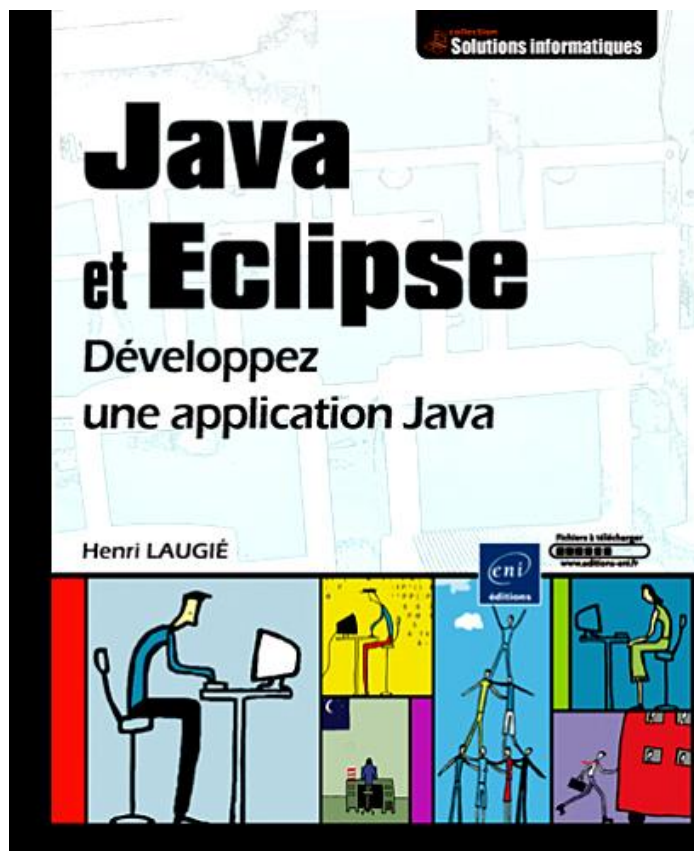
www.editions-eni.fr



Java et Eclipse

Développez une application Java

Henri LAUGIÉ



Résumé

Ce livre sur **Java** et **Eclipse** s'adresse aux développeurs, étudiants en informatique et élèves ingénieurs. Il permet au lecteur de maîtriser Java SE 6 ainsi que l'environnement de développement intégré Eclipse à travers le développement d'une application de gestion.

Vous découvrirez comment construire rapidement des interfaces graphiques avec **Eclipse et Visual Editor**, comment contrôler la souris, le clavier, comment **gérer les événements** en maîtrisant les écouteurs et les adaptateurs, comment exploiter une base de données **MySQL avec JDBC**, afficher les données en mode fiche ou table avec le composant **JTable** et comment créer avec **Java et SQL** les principales fonctionnalités d'ajout, de suppression, de modification et de recherche.

Vous apprendrez également à bâtir votre application selon une **approche Génie logiciel** et vous vous familiariserez avec les **diagrammes d'UML2** en utilisant **Eclipse UML Free Edition**. Vous apprendrez aussi à structurer votre code selon le **modèle MVC**.

En privilégiant l'apprentissage par la pratique, l'auteur va à l'essentiel et prend soin d'expliquer le plus clairement possible les notions complexes rencontrées au cours du développement.

Les exemples cités dans l'ouvrage sont en téléchargement sur le site de l'éditeur.

L'auteur

A la fois formateur, ingénieur et professeur d'informatique, **Henri Laugié** allie compétences et expérience aussi bien techniques que pédagogiques. Dans cet ouvrage, privilégiant l'apprentissage par la pratique, l'auteur va à l'essentiel et prend soin d'expliquer le plus clairement possible les notions complexes rencontrées au cours du développement.

Ce livre numérique a été conçu et est diffusé dans le respect des droits d'auteur. Toutes les marques citées ont été déposées par leur éditeur respectif. La loi du 11 Mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayant cause, est illicite" (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal.
Copyright Editions ENI

Installation du JDK

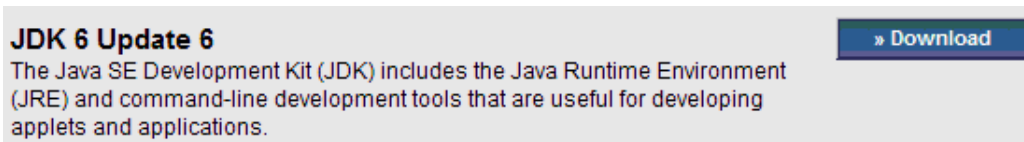
Vérifiez auparavant la présence et la version de Java. Sous Windows (2000, XP, Vista), vous pouvez le faire rapidement par **Démarrer - Exécuter**.

- Dans la fenêtre, tapez la commande **cmd** puis validez.
- La fenêtre de l'interpréteur de commandes apparaît. Tapez la commande : **java -version**

```
C:\Documents and Settings\Mon PC>java -version
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode, sharing)
```

Dans la copie d'écran, **java version "1.6.0_03"** signifie que la version 6 de Java est installée. Les projets présentés dans cet ouvrage nécessitent la présence de cette version. Pour une installation ou une mise à jour, vous pouvez vous rendre sur différents sites. En voici deux dont celui de Sun Microsystems :

- Sun : <http://www.sun.com/download/index.jsp>



- Java.com : <http://www.java.com/fr/download/manual.jsp>

 [Windows XP/Vista/2000/2003 Hors ligne](#) * taille du fichier : 15.18 MB

Choisissez la version SE, Standard Edition, qui correspond à votre système d'exploitation : Windows, Linux ou Solaris (évitez les versions bêta).



Le JDK, Java Development Kit, également nommé J2SE contient la machine virtuelle J2RE, Java 2 Runtime Environnement.

- Procédez à l'installation.

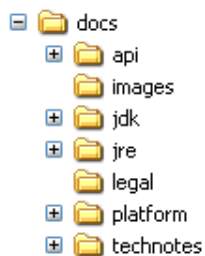


Il est indispensable de disposer de l'aide.

- Téléchargez celle-ci sur le site de Sun Microsystems.



- Procédez à l'installation en choisissant un emplacement sur votre disque dur. L'aide est rangée par défaut dans un dossier nommé **docs**.



- Pour accéder à l'aide, cliquez sur l'icône **index** se trouvant dans le dossier **docs**. La création d'un raccourci vous permettra de gagner du temps.



Installation d'Eclipse

- Téléchargez sur le site d'Eclipse et installez la version **Classic 3.2.** ou une version plus récente (évituez les versions bêta) :

<http://www.eclipse.org/downloads/>

- Décompressez le fichier téléchargé dans un dossier de votre choix puis lancez Eclipse sous Windows en double cliquant sur le fichier **eclipse.exe**. La création d'un raccourci vous permettra de gagner du temps.



Si l'écran de démarrage reste affiché et que l'application ne se lance pas, c'est que celle-ci n'arrive pas à trouver le JDK. Recherchez l'emplacement de celui-ci sur le disque dur. Normalement en C:, l'installation du JDK se fait dans les dossiers suivants : **C:\Sun\AppServer\jdk**.

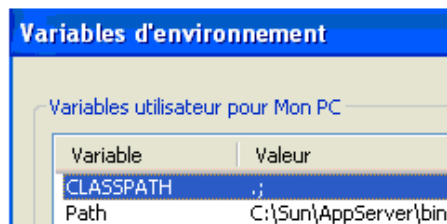
- en ligne de commandes avec **Edit** ou sous Windows avec **Bloc Notes** :

SET CLASSPATH=.;C:\Sun\AppServer\jdk\lib

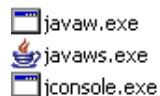
N'oubliez pas le point, il indique le répertoire courant.

- sous Windows par le **Panneau de configuration** :

Cliquez sur l'icône **Système**, sur l'onglet **Avancé** puis sur le bouton **Variables d'environnement**. Créez la variable CLASSPATH et ajoutez au chemin **Path**, la ligne ci-dessous.

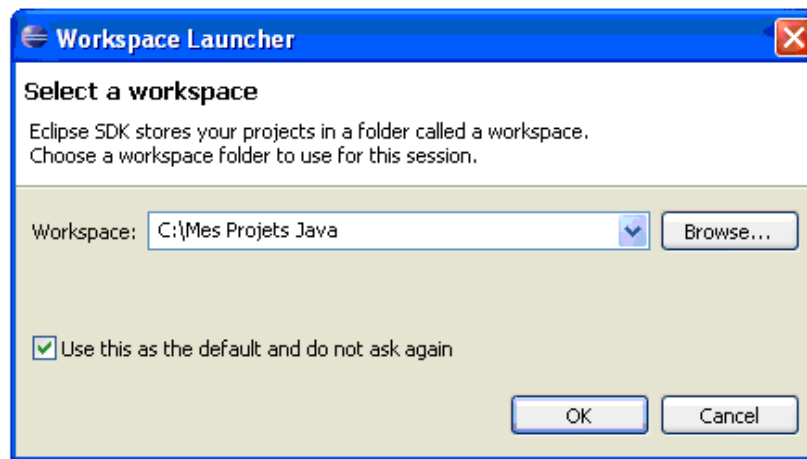



Le dossier **bin** contient les programmes exécutables du SDK. Voici un extrait du contenu du dossier :



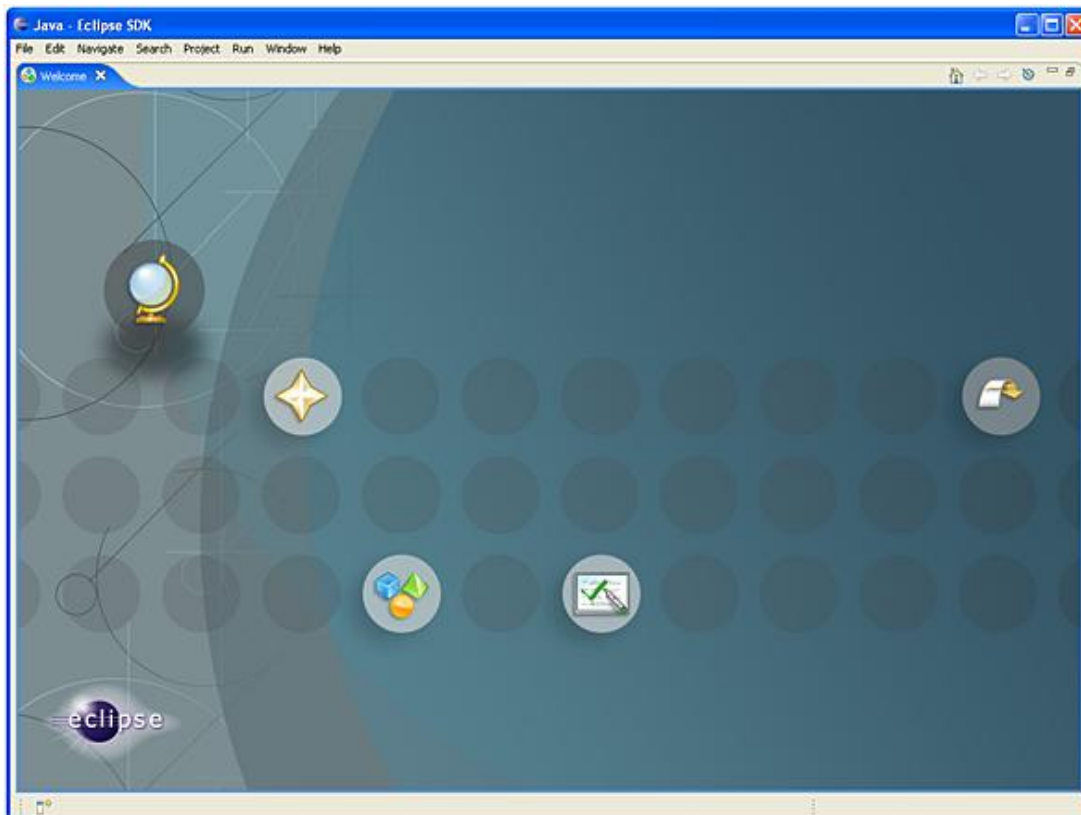
Vous devez redémarrer votre ordinateur pour que les modifications soient prises en compte.

Lors du premier lancement, Eclipse propose un dossier par défaut nommé **workspace**. Créez et sélectionnez plutôt un autre dossier plus personnel, par exemple : C:\Mes Projets Java. Cochez la case **Use this as the default and do not ask again**.



 Toutes les copies écran concernant Eclipse correspondent à la version 3.2.

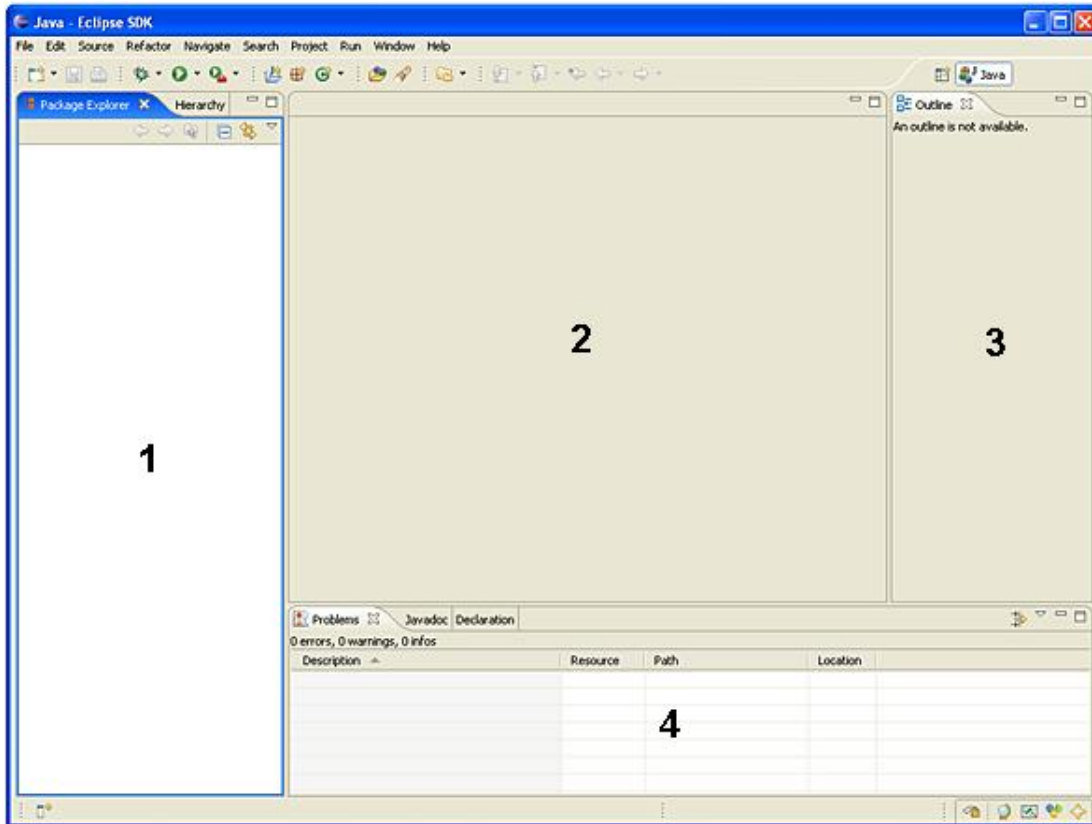
Eclipse présente ensuite une page d'accueil permettant d'obtenir des informations sur l'EDI (*Environnement de Développement Intégré*) réparties en quatre thèmes.



- **overview** : permet d'accéder rapidement à la partie de l'aide en ligne correspondant au thème sélectionné.
- **tutorials** : permet d'accéder à des assistants qui permettent, sous la forme de didacticiels, de réaliser de simples applications ou plugins.
- **samples** : permet de lancer des exemples d'applications à charger sur Internet.
- **what's new** : permet d'accéder rapidement à la partie de l'aide en ligne concernant les nouveautés d'Eclipse.

Découverte de l'IDE

Nous allons sans plus tarder accéder au **workbench** ou plan de travail. Cliquez sur la flèche.



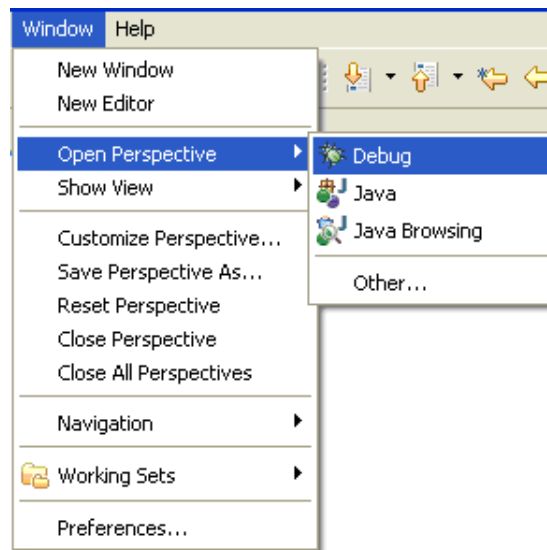
Eclipse propose un environnement de développement par défaut nommé **perspective** composé de quatre fenêtres. Plus précisément, il s'agit de la perspective Java. Il existe d'autres perspectives qui sont utilisées selon les besoins des développeurs.

Chaque perspective est constituée d'un certain nombre d'éléments appelées **views** (vues) qui ne sont pas toutes forcément ouvertes (visibles). La perspective Java présente à l'ouverture est composée des vues suivantes :

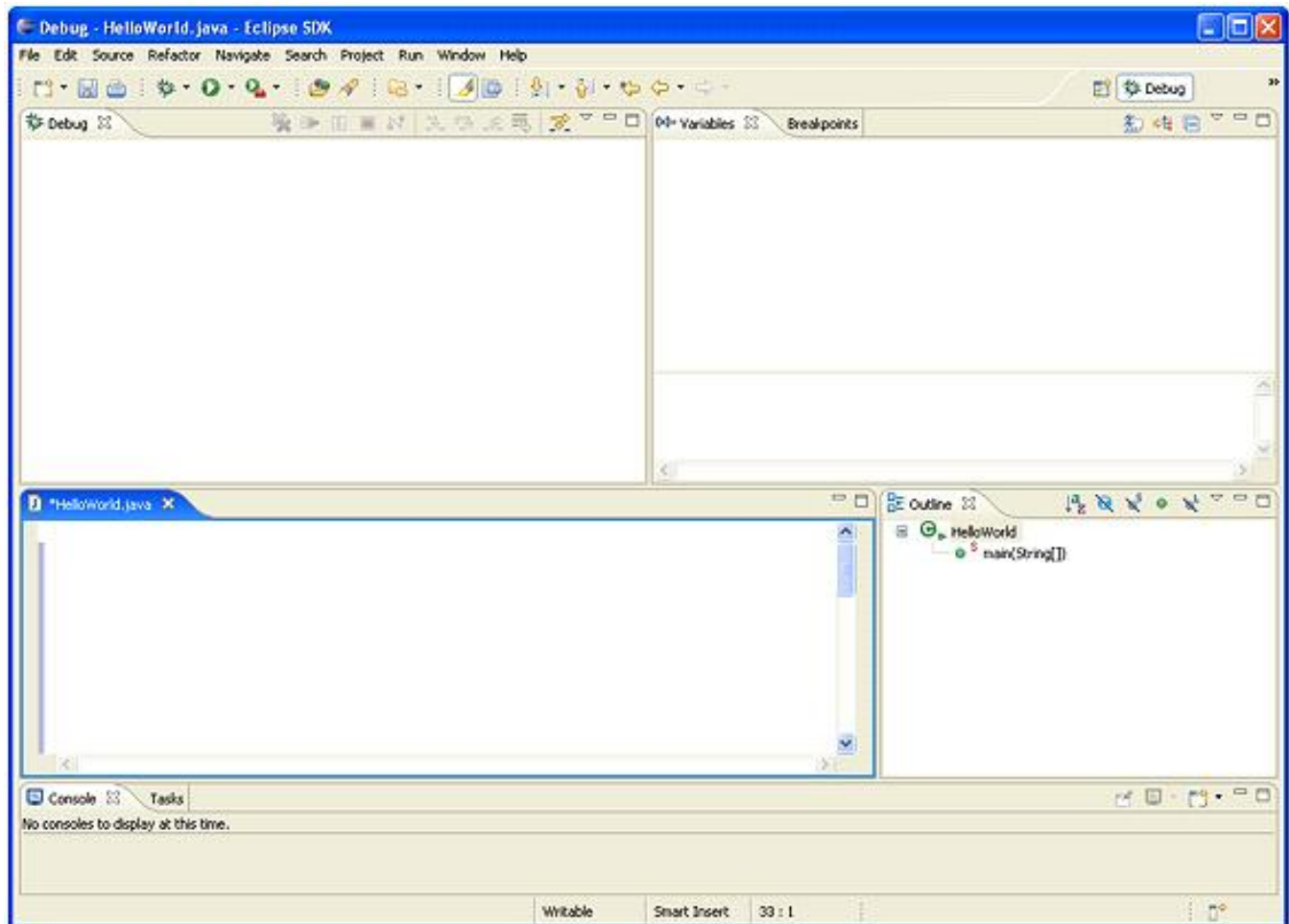
- 1 : explorateur de packages
- 2 : éditeur de code
- 3 : structure
- 4 : plusieurs vues dans une même fenêtre accessibles par des onglets


Il est possible de personnaliser une perspective en ouvrant ou en refermant des vues. Nous allons effectuer quelques essais.

- Ouvrez la perspective **Debug**.
















L'environnement de développement est totalement modifié et convient maintenant aux tâches de débogage.



- Revenez à la perspective Java en l'ouvrant par le menu **Window - Open Perspective Java**. Vous pouvez aussi cliquer sur l'icône  située dans le coin supérieur droit de la fenêtre d'Eclipse.

Eclipse dispose de nombreuses vues.

- Pour ajouter par exemple la vue Console à la perspective ouverte, choisissez dans le menu **Window - Show View - Console**.

 Ant	
 Console	Alt+Shift+Q, C
 Declaration	Alt+Shift+Q, D
 Error Log	
 Hierarchy	Alt+Shift+Q, T
 Javadoc	Alt+Shift+Q, J
 Navigator	
 Outline	
 Package Explorer	Alt+Shift+Q, P
 Problems	
 Progress	
 Search	Alt+Shift+Q, S
 Tasks	
<hr/>	
Other...	Alt+Shift+Q, Q

Ajout de plugins

Nous allons personnaliser Eclipse en ajoutant plusieurs plugins pour mener à bien notre projet.

1. Traductions

Par défaut Eclipse est en anglais. I.B.M. propose des traductions pour les versions 3.0.x d'Eclipse dans différentes langues.

- Téléchargez le fichier à l'adresse suivante :

<http://download.eclipse.org/eclipse/downloads/>


Language Packs	
Build Name	Build Date
3.2_Language_Packs	Wed, 12 Jul 2006 -- 17:00 (-0400)
3.2.1_Language_Packs	Thu, 21 Sep 2006 -- 09:45 (-0400)

- Créez sur votre disque dur un dossier provisoire, par exemple : Langues Eclipse Java. Décompressez tous les fichiers dans ce dossier.

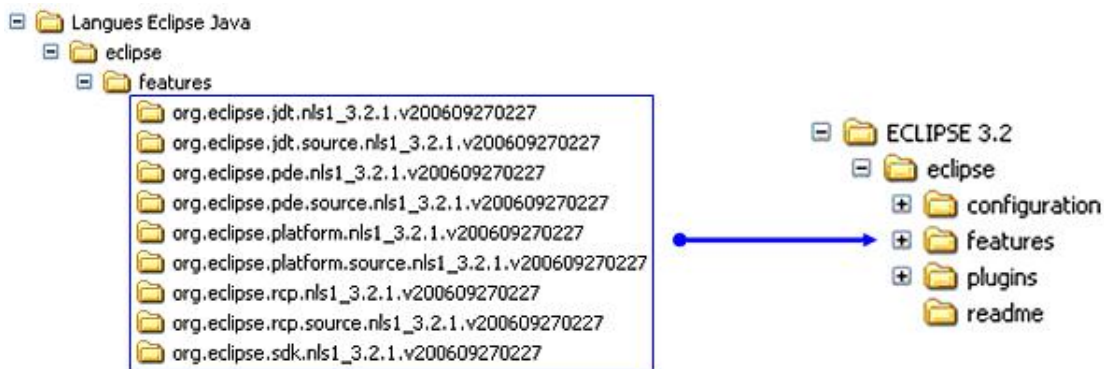
L'extraction crée le dossier **eclipse** contenant les sous-dossiers **features** et **plugins** qui contiennent eux-mêmes de nombreux dossiers.



Il faut ensuite ajouter le contenu et seulement le contenu des sous-dossiers features et plugins dans les sous-dossiers de même nom du dossier d'installation d'Eclipse.

 Attention, dans le dossier d'Eclipse, deux dossiers aux noms de features et plugins existent déjà. Ne les "écrasez" pas ! Sinon il vous faudra tout réinstaller.

- Recopiez seulement les contenus dans les sous-dossiers correspondants du dossier d'Eclipse. Exemple pour le sous-dossier features :



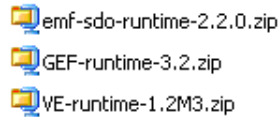
Les traductions seront prises en compte au prochain démarrage d'Eclipse.

- Quittez Eclipse puis relancez-le. Si vous n'avez pas fait d'erreurs de copie, les menus sont désormais en français.

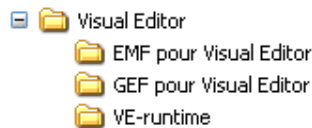
2. Visual Editor

Visual Editor permet de gagner un temps considérable dans l'élaboration des IHM, interfaces homme-machine. Il est cependant utile de comprendre et de savoir développer des applications graphiques avec Swing.

Pour utiliser Visual Editor, il faut télécharger trois fichiers et les installer dans l'ordre suivant :




- Quittez auparavant Eclipse.
- Téléchargez les fichiers sur le site d'Eclipse à l'adresse ci-après :
<http://www.eclipse.org/vep/>
ou plus directement à l'adresse suivante :
<http://download.eclipse.org/tools/ve/downloads/drops/R-1.2-200606280938/index.html>
- Dézippez les fichiers dans un dossier nommé par exemple "Visual Editor". L'extraction crée trois dossiers :



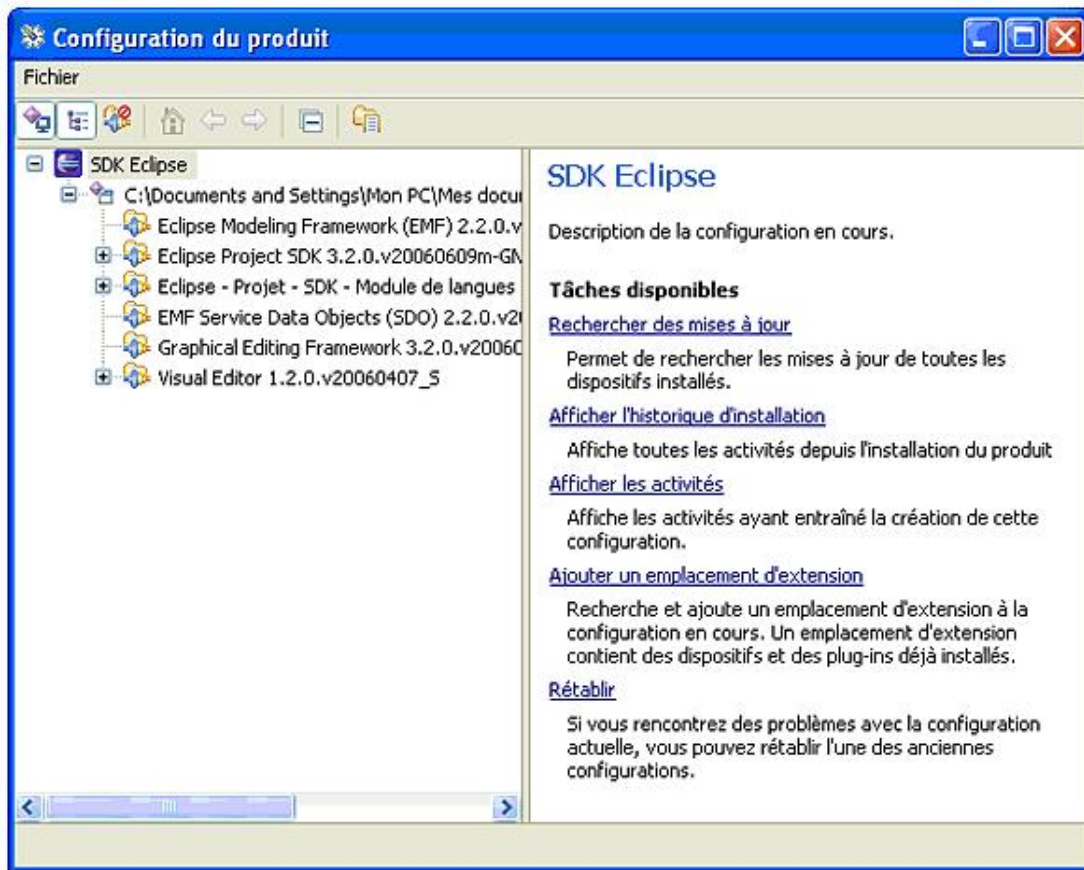
Pour chacun de ces trois dossiers, vous devez suivre la démarche suivante :

- ouvrir le dossier
- décompresser le fichier zip
- recopier comme vu précédemment, uniquement le contenu des sous-dossiers features et plugins dans les sous-dossiers correspondants du dossier d'installation d'Eclipse en respectant l'ordre suivant : EMF, GEF puis VE.

 L'extraction du fichier compressé du dossier VE-runtime génère le plugin Visual Editor Project qui a besoin pour fonctionner des ressources contenues par les deux autres plugins issus des dossiers GEF et EMF. Le plugin GEF permet de créer les interfaces graphiques et le plugin GEF génère le code correspondant. Le lien établi est bidirectionnel, toute modification sur un composant met à jour le code et réciproquement.

Nous allons vérifier que toutes les ressources ont été bien installées.

- Choisissez dans le menu **Aide - Mise à jour des logiciels - Gérer la configuration**.



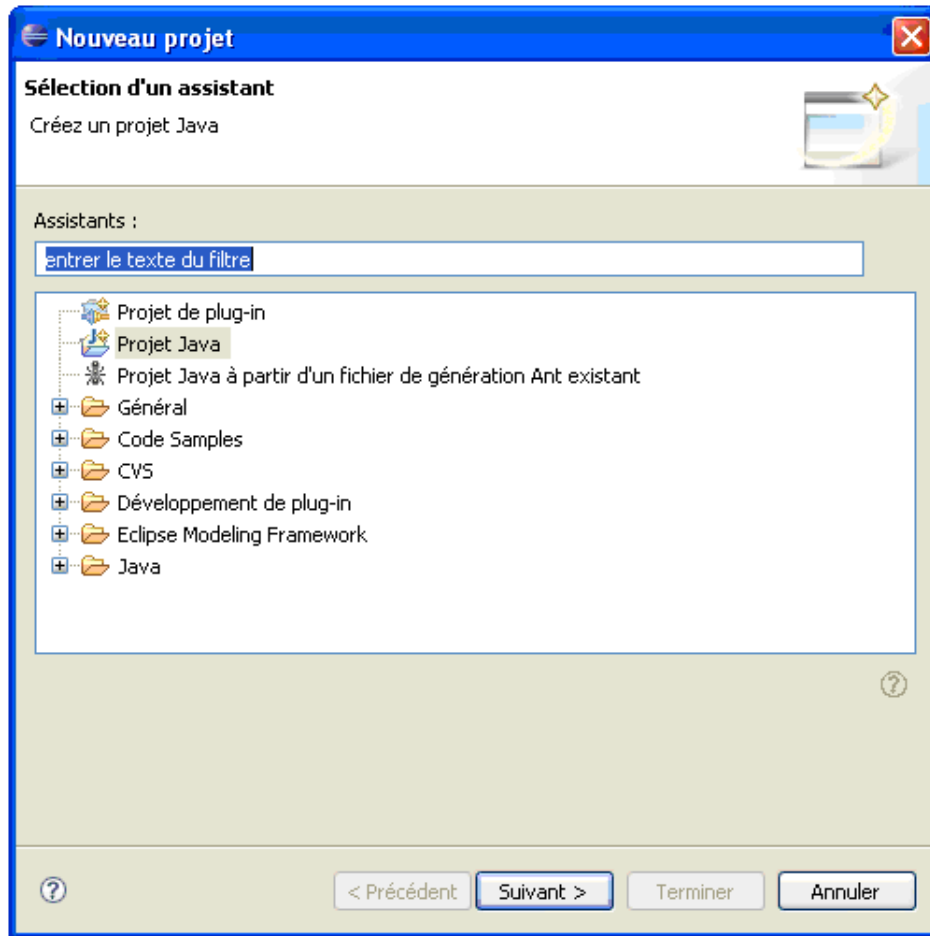
La création d'interfaces graphiques avec Visual Editor est désormais possible. Vérifiez qu'il est bien présent dans le menu en cliquant sur **Fichier - Nouveau**. L'option **Visual Class** doit être présente.



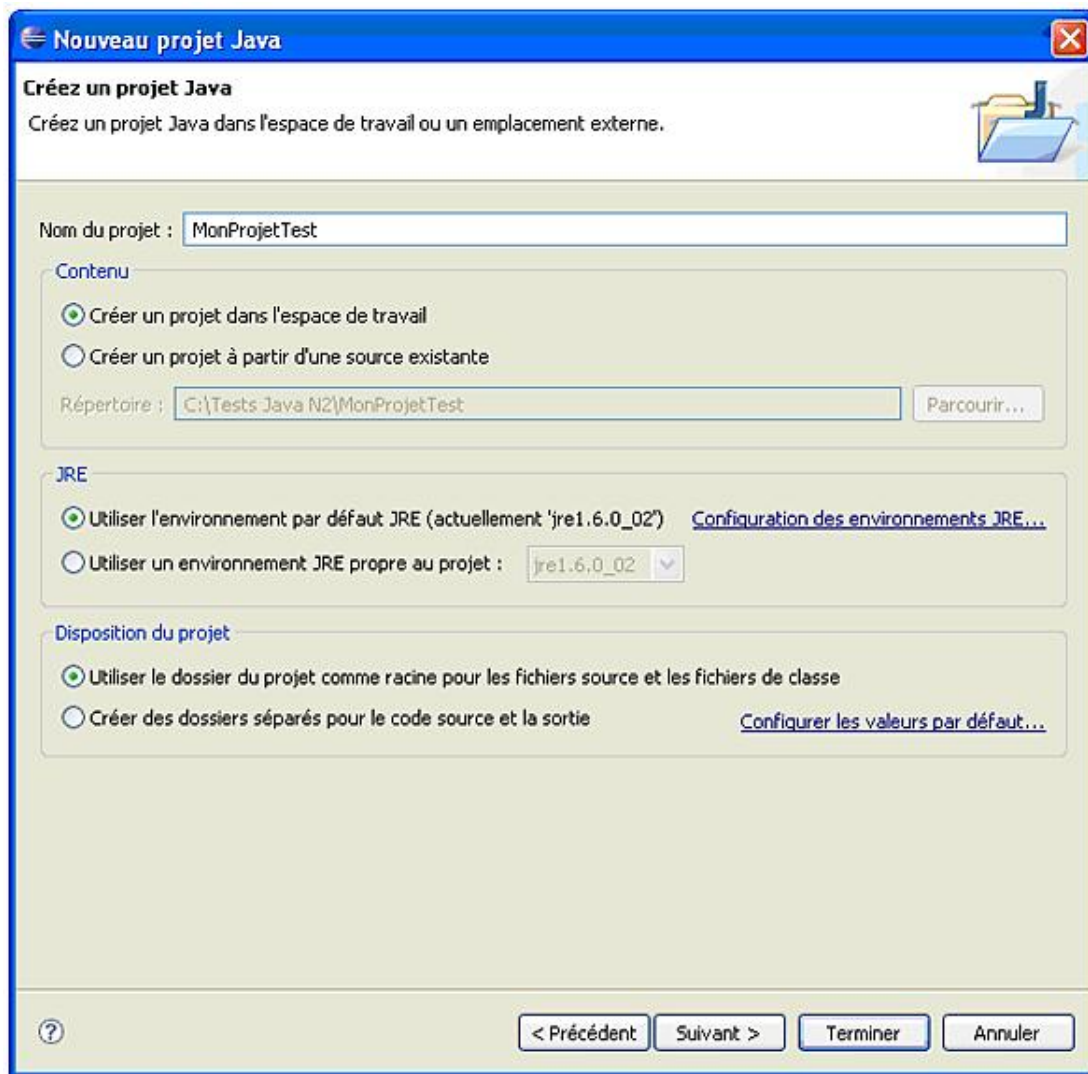
Première classe

Pour nous familiariser avec l'**espace de travail**, nous allons créer une classe très simple. Il nous faut auparavant commencer par créer un projet puis un paquetage, toute classe devant être rangée dans un paquetage (ou package).

- Dans le menu, choisissez **Fichier - Nouveau - Projet**.



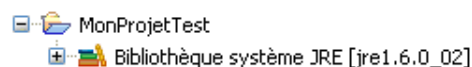
- Ne saisissez rien. Cliquez sur le bouton **Suivant**.



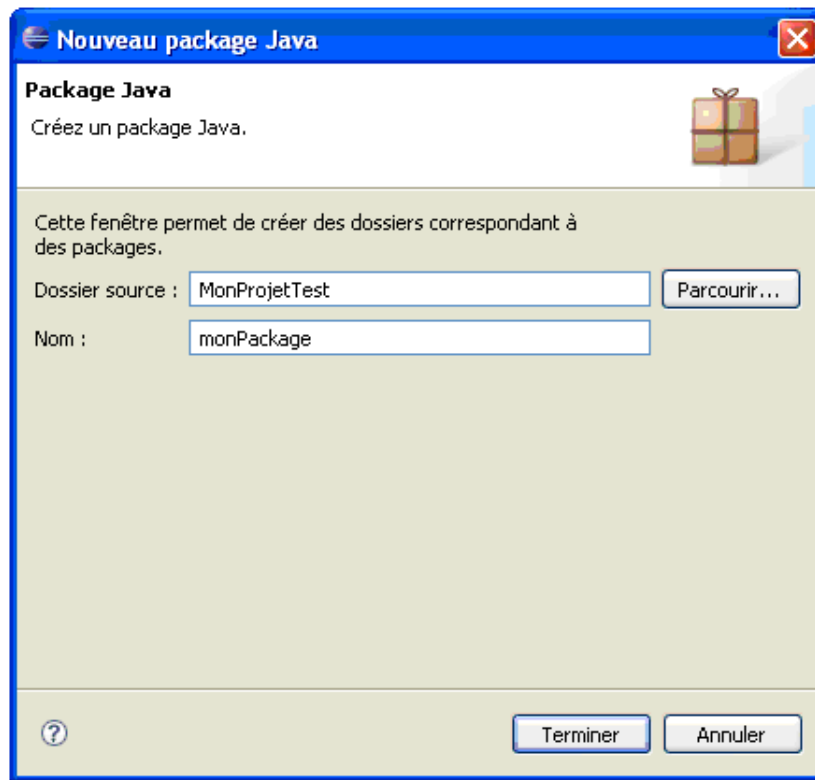
- Nommez le projet **MonProjetTest**.

Le projet sera créé dans le dossier que vous avez choisi comme espace de travail lors de l'installation d'Eclipse. Vous pouvez cependant en choisir en autre. Il est aussi possible de changer de JRE si plusieurs machines virtuelles Java sont installées sur le poste.

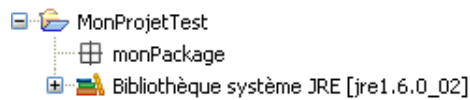
- Cliquez sur le bouton **Terminer**. Votre projet et son contenu sont désormais visibles dans l'explorateur de paquets.



- Sélectionnez votre projet dans l'explorateur, puis choisissez dans le menu, **Fichier - Nouveau - Package**. Nommez-le **monPackage** en commençant par une minuscule pour respecter les conventions d'écriture du langage Java puis cliquez sur le bouton **Terminer**.



Votre paquetage est ajouté au projet. Nous pouvons maintenant créer une classe.



Avant de poursuivre, allons voir les dossiers créés sur le disque dur. Vous devriez obtenir une structure semblable à celle-ci :



- Votre projet étant sélectionné, choisissez dans le menu **Fichier - Nouveau - Classe**.
- Nommez la classe **MonPremierProgramme**, en commençant par une majuscule :
 - Cochez **public static void main(String[] args)**.
 - Cliquez sur le bouton **Terminer**.

Nouvelle classe Java

Classe Java
Créez une classe Java

Dossier source :

Package :

☐ Type englobant :

Nom :

Modificateurs : ☒ public ☐ Valeur par défaut ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclasse :

Interfaces :

Quels raccords de méthode voulez-vous créer ?

☒ public static void main(String[] args)

☐ Constructeurs de la superclasse

☒ Méthodes abstraites héritées

Voulez-vous ajouter des commentaires conformément à la configuration spécifiée dans les [propriétés](#) du projet sélectionné ?

☐ Générer les commentaires

La classe a été ajoutée au paquetage et le code généré automatiquement par Eclipse est visible dans l'éditeur de code.

```
*MonPremierProgramme.java x
package monPackage;

public class MonPremierProgramme {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Raccord de méthode auto-généré
    }

}
```

- Nettoyez le code et ajoutez la ligne suivante :

```
package monPackage;

public class MonPremierProgramme {
    public static void main(String[] args) {
        System.out.println("Java, c'est extra !");
    }
}
```

Voilà, nous y sommes. De nombreuses notions propres à la POO, Programmation Orientée Objet, sont déjà présentes dans ces quelques lignes. Pour ceux qui découvrent Java, voici une présentation sommaire. Nous reverrons toutes ces notions lors de la réalisation de notre projet.



Cette classe comporte un bloc principal avec un bloc interne. Elle porte le même nom que le fichier dans lequel elle est sauvegardée.

Les blocs sont délimités par des accolades { ... }.

L'ouverture du bloc se fait avec une accolade en fin de ligne.

Par convention, la fermeture du code se fait sur une ligne isolée (sauf si le bloc ne contient qu'une seule ligne).

1^{ère} ligne

Les instructions **public class** indiquent que cette classe est publique et peut être utilisée par n'importe quelle autre classe.

2^{ème} ligne

Le mot clé **public** pour `public static void main(...)` indique que cette méthode peut être utilisée par n'importe quelle autre classe.

Le mot clé **static** indique qu'il s'agit d'une méthode statique ou de classe (on dit aussi à portée de classe). Cette méthode ne peut être héritée. Elle appartient uniquement à cette classe bien qu'elle soit utilisée par toutes ses instances.

Le mot clé **void** signifie rien. Il est utilisé pour désigner des méthodes qui réalisent des actions sans retourner un résultat. Ces méthodes correspondent aux procédures dans un langage non objet.

void main(...) désigne la méthode principale du programme, « la porte d'entrée », et précise qu'elle ne retourne rien.

La méthode **void main()** prend un paramètre obligatoire nommé **argv** de type **String[]**. Par convention, le terme **argv** est employé mais on peut en choisir un autre.

3^{ème} ligne

Cette ligne constitue le corps du programme.

System désigne une classe Java dont la classe parente est **Object**. En consultant l'aide, vous pouvez visualiser la hiérarchie des classes.

- Double cliquez sur le fichier **index.htm** dans le dossier où vous avez installé l'aide ou sur le raccourci que vous avez créé (voir le chapitre Environnement de développement).
- Dans le volet gauche, choisissez le paquetage **java.lang** puis la classe **System**.



Vous obtenez la hiérarchie suivante et toute l'aide concernant la classe **System**.

```

java.lang.Object
└─ java.lang.System

```

Ces deux classes sont rangées dans le paquetage **lang**, lui-même contenu dans le paquetage **java**.

out désigne une **propriété** de la classe **System**, **out** étant elle-même du type classe **PrintStream** dont la hiérarchie avec les classes parentes est présentée ci-après :

```

java.lang.Object
└─ java.io.OutputStream
    └─ java.io.FilterOutputStream
        └─ java.io.PrintStream

```

En consultant la classe **PrintStream**, on constate que **println** est une méthode de cette classe. Cette méthode est utilisable dans la classe **System** via la propriété **out**. La méthode **println** affiche la valeur littérale qui se trouve encadrée par des guillemets et effectue un retour à la ligne.

Nous retiendrons à ce stade les points suivants :

- Un programme comporte au moins une classe.
- Les classes doivent porter le même nom que les fichiers dans lesquels elles se trouvent.
- Une seule classe doit posséder la méthode principale : **main()**. C'est toujours elle qui est exécutée en premier.
- Toutes les méthodes proviennent obligatoirement d'une classe (ou d'une interface) - classe Java ou classe créée

par le développeur.

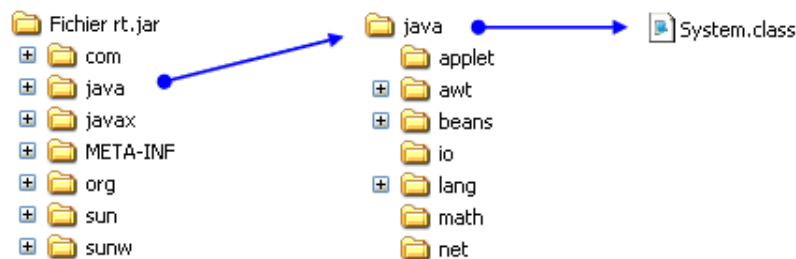
- Les classes sont regroupées dans des packages ou paquetages. Les paquetages peuvent contenir d'autres paquetages.
- Java est sensible à la casse. Les minuscules et les majuscules sont des caractères différents.

 Les paquetages Java et leurs classes sont stockés dans le fichier **rt.jar** qui est un fichier compressé. Pour une installation sur C:, le chemin peut être : **C:\Sun\AppServer\jdk\jre\lib**

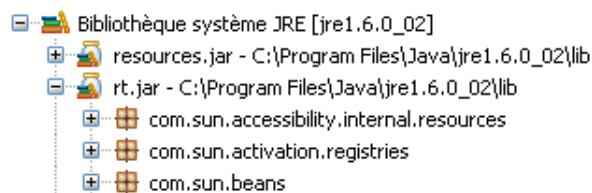


Si vous voulez mieux comprendre l'organisation des paquetages et de leurs classes sur le disque dur, vous pouvez les visualiser en recopiant ce fichier dans un dossier nommé par exemple Fichier rt.jar et le décompresser.

Voici des extraits :



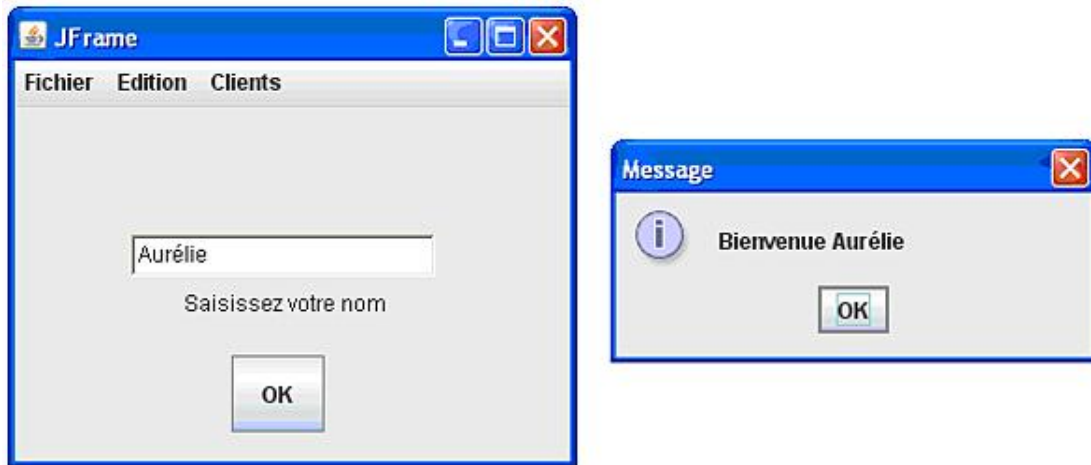
Sous Eclipse, vous pouvez consulter dans l'explorateur de paquetages, la bibliothèque système complète du JRE.



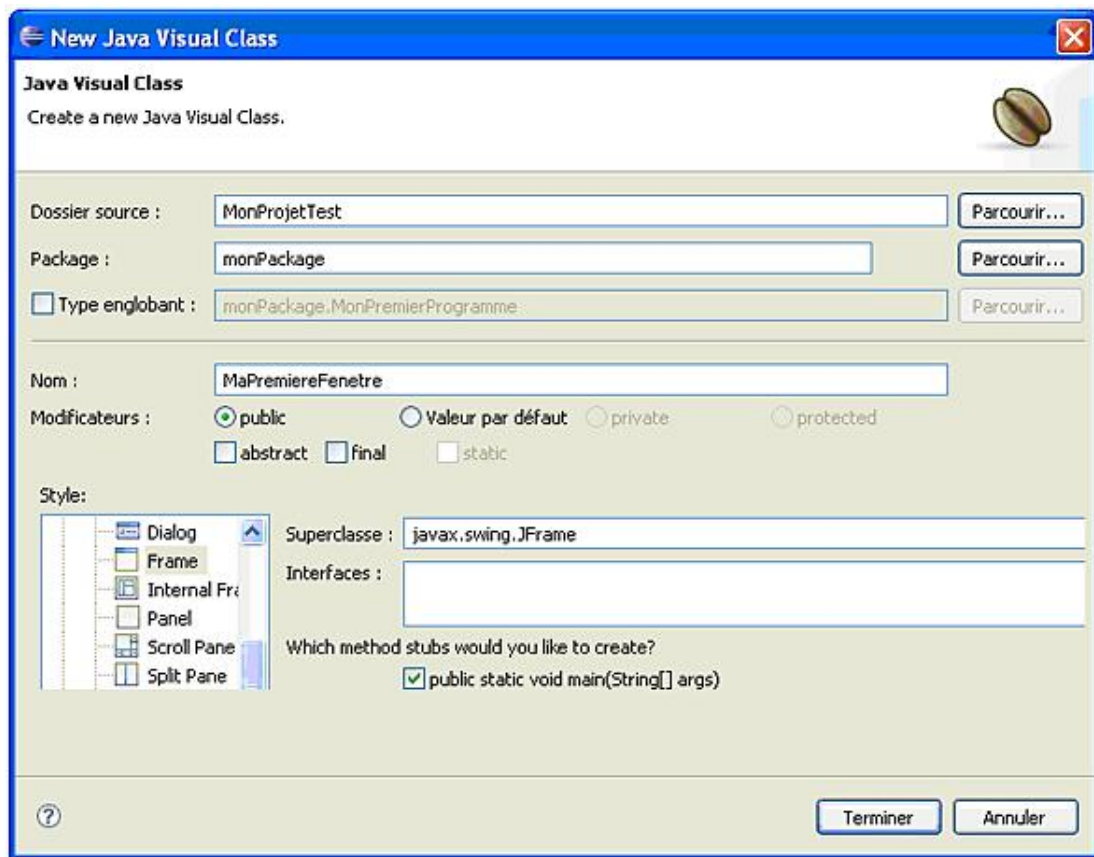
Première fenêtre

1. Création de la fenêtre

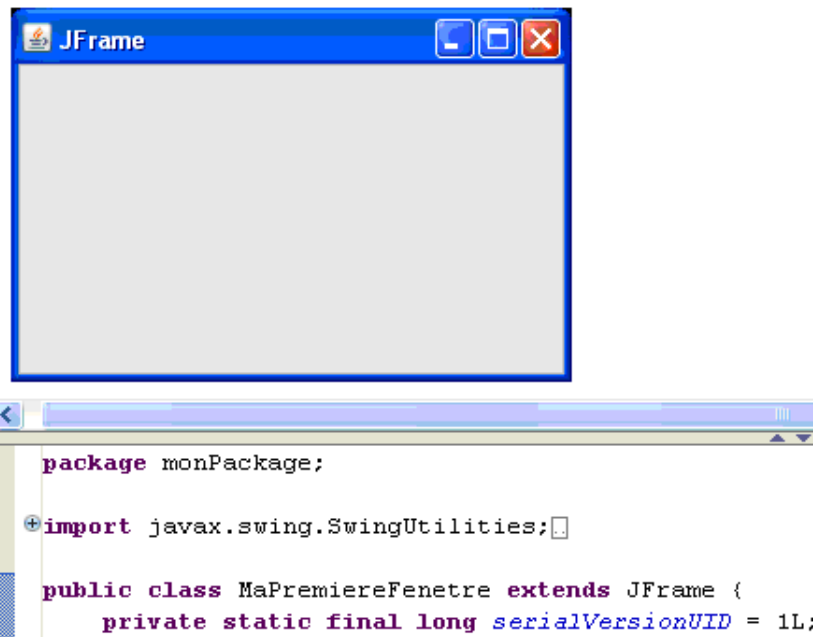
Nous allons créer une simple fenêtre en nous aidant de Visual Editor installé précédemment. Celle-ci comportera un champ de saisie dans lequel l'utilisateur pourra saisir un nom. Un clic sur le bouton **OK**, lui présentera un message comportant ce nom.



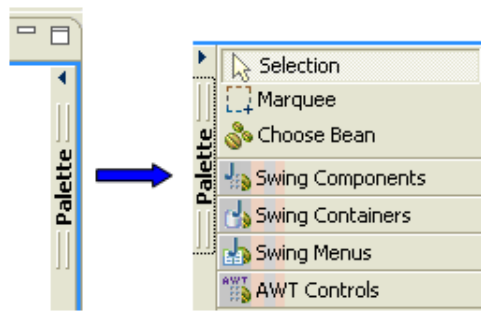
- Sélectionnez le projet, et choisissez **Fichier - Nouveau - Visual Class**.
 - Nommez cette classe graphique **MaPremiereFenetre**.
 - Sélectionnez dans **Style** l'élément **Frame**.
 - Cochez **public static void main(String[] args)** puis cliquez sur le bouton **Terminer**.



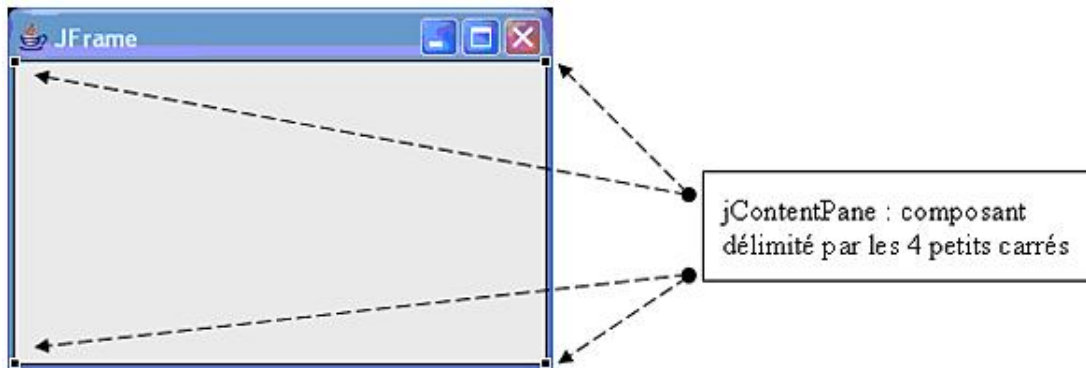
La fenêtre apparaît au-dessus du code qui a été généré automatiquement.



La palette se trouvant à droite de la fenêtre contient de nombreux composants pour la création d'interfaces graphiques. Ceux-ci sont regroupés en bibliothèques. Pour la faire apparaître déplacez la souris sur la bande comportant le mot **Palette**.



La fenêtre créée contient un composant particulier nommé **jContentPane** dans lequel on peut déposer d'autres composants tels que des boutons ou des champs de saisie.



Code généré :

```
private JPanel getJContentPane() {
    if (jContentPane == null) {
        jContentPane = new JPanel();
        jContentPane.setLayout(new BorderLayout());
    }
    return jContentPane;
}
```

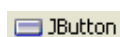
2. Ajout d'un bouton

- Cliquez sur le composant **jContentPane**.

Pour pouvoir placer librement un bouton ou tout autre composant, mettez la propriété **>layout** à **null**. À noter que les composants ne seront alors plus automatiquement redimensionnés si la taille de la **Frame** est modifiée.

Property	Value
background	238,238,238
border	
componentOrientation	UNKNOWN
enabled	true
>field name	jContentPane
font	Dialog, plain, 12
foreground	51,51,51
>layout	null

- Dans la palette, cliquez sur le composant **JButton**.



- Dessinez un bouton dans la fenêtre et redimensionnez-le. Changez le libellé en cliquant une seconde fois dessus

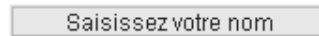
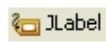
après sélection du bouton.



Procédez de la même manière que pour le bouton pour ajouter un champ de saisie ou un libellé à la fenêtre.



Ajout d'une zone de saisie



Ajout d'un libellé

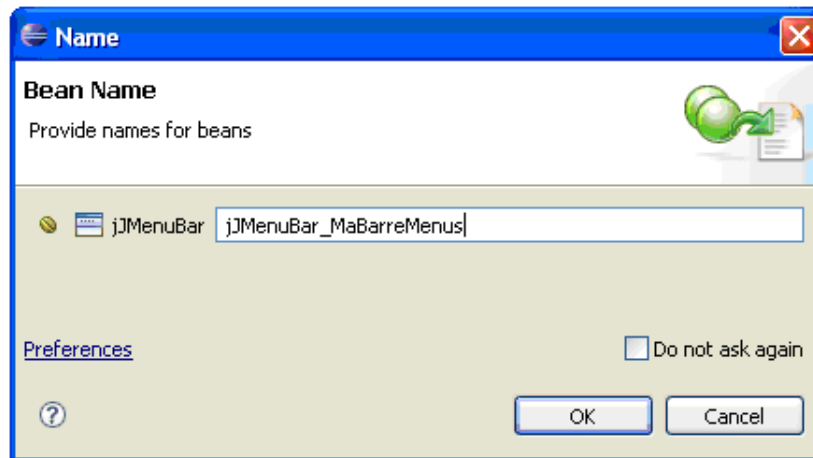
Création d'un menu



- Cliquez sur la barre de titre du cadre.



- Une fenêtre apparaît. Validez.



- Au bas de l'écran, modifiez la largeur de la barre de menu en changeant la valeur 0,2 par 0,25. Attention, il ne s'agit pas d'une valeur décimale mais de deux valeurs différentes séparées par une virgule.

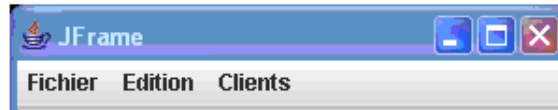
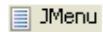
Property	Value
enabled	true
>field name	jMenuBar
font	Dialog, bold, 12
foreground	51,51,51
name	
>preferredSize	0,25
toolTipText	
visible	true

Le code correspondant au menu est généré et les modifications réalisées dans la fenêtre **Properties** sont automatiquement répercutées dans le code.

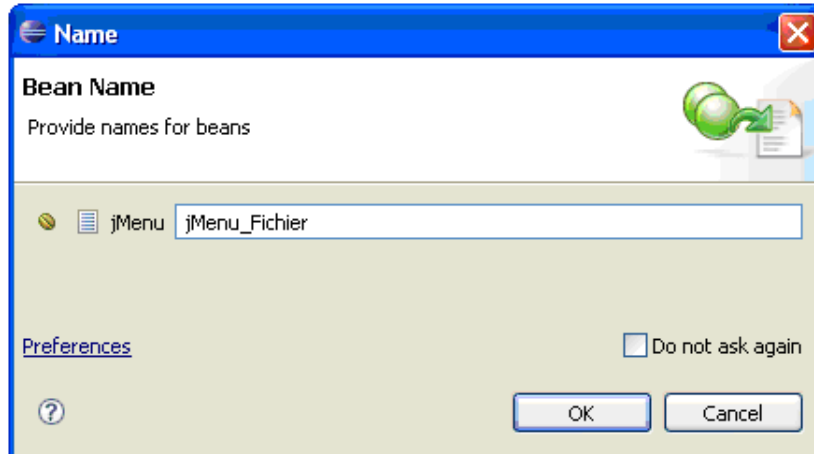
```
private JMenuBar getJJMenuBar_MaBarreMenus() {  
    if (jMenuBar_MaBarreMenus == null) {  
        jMenuBar_MaBarreMenus = new JMenuBar();  
        jMenuBar_MaBarreMenus.setPreferredSize(new Dimension(0, 25));  
    }  
    return jMenuBar_MaBarreMenus;  
}
```

1. Ajout des menus à la barre de menus

Nous allons créer la barre de menus suivante avec le composant **JMenu**.



- Pointez la barre de menu. Une fenêtre apparaît. Validez.



- Le premier menu est déposé à gauche de la barre. Cliquez dessus pour changer son libellé.



- Procédez de la même manière pour les menus **Edition** et **Clients**.

Code de la barre de menus suite à l'ajout des trois menus :

```
private JMenuBar getJJMenuBar_MaBarreMenus() {
    if (jJMenuBar_MaBarreMenus == null) {
        jJMenuBar_MaBarreMenus = new JMenuBar();
        jJMenuBar_MaBarreMenus.setPreferredSize(new Dimension(0, 25));
        jJMenuBar_MaBarreMenus.add(getJMenu_Fichier());
        jJMenuBar_MaBarreMenus.add(getJMenu_Edition());
        jJMenuBar_MaBarreMenus.add(getJMenu_Clients());
    }
    return jJMenuBar_MaBarreMenus;
}
```

Code des menus :

```
private JMenu getJMenu_Fichier() {
    if (jMenu_Fichier == null) {
        jMenu_Fichier = new JMenu();
        jMenu_Fichier.setText("Fichier");
    }
    return jMenu_Fichier;
}
private JMenu getJMenu_Edition() {
    if (jMenu_Edition == null) {
        jMenu_Edition = new JMenu();
        jMenu_Edition.setText("Edition");
    }
}
```

```

    return jMenu_Edition;
}
private JMenu getJMenu_Clients() {
    if (jMenu_Clients == null) {
        jMenu_Clients = new JMenu();
        jMenu_Clients.setText("Clients");
    }
    return jMenu_Clients;
}
}

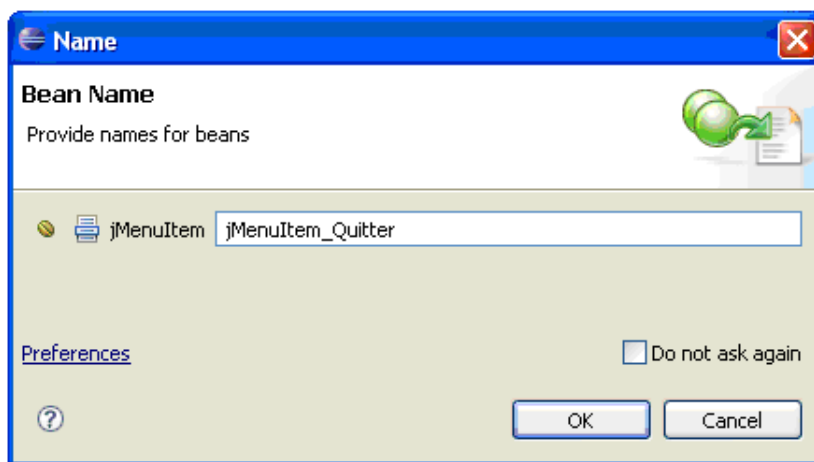
```

2. Ajout des options aux menus

- Utilisez le composant **JMenuItem**.



- Faites glisser un composant JMenuItem sur le menu **Fichier**.



Le code du menu **Fichier** s'est enrichi d'une ligne.

```

private JMenu getJMenu_Fichier() {
    ...
    jMenu_Fichier.add(getJMenuItem_Quitter());
    ...
}

```

Et le code concernant l'option **Quitter** a été créé.

```

private JMenuItem getJMenuItem_Quitter() {
    if (jMenuItem_Quitter == null) {
        jMenuItem_Quitter = new JMenuItem();
    }
    return jMenuItem_Quitter;
}


```

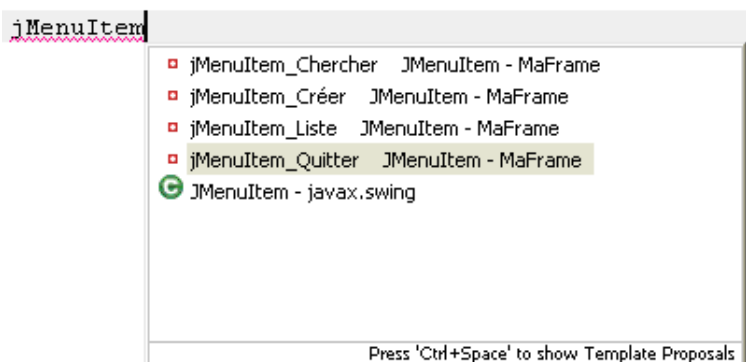
- Rajoutez vous-même dans le code une ligne pour que l'option ait un libellé.

```

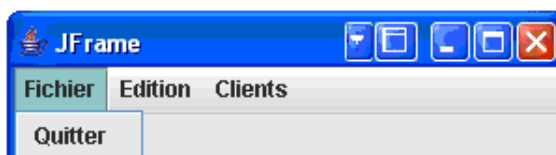
private JMenuItem getJMenuItem_Quitter() {
    ...
    jMenuItem_Quitter.setText("Quitter");
    ...
}

```

 **Auto-complétion.** L'auto-complétion permet de compléter automatiquement vos lignes de code dès lors que le terme à compléter existe dans le lexique de Java Eclipse. Pour bénéficier de l'auto-complétion de code appuyez simultanément sur les touches [Ctrl] + [Espace] de votre clavier. Une liste d'éléments commençant par le dernier mot saisi apparaît. Il ne reste plus qu'à sélectionner l'élément désiré et à valider.

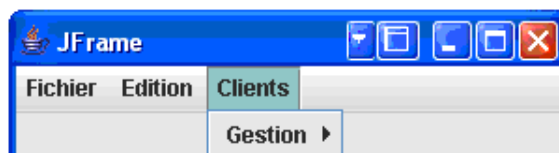


- Renouvelez les opérations précédentes pour obtenir les menus suivants :



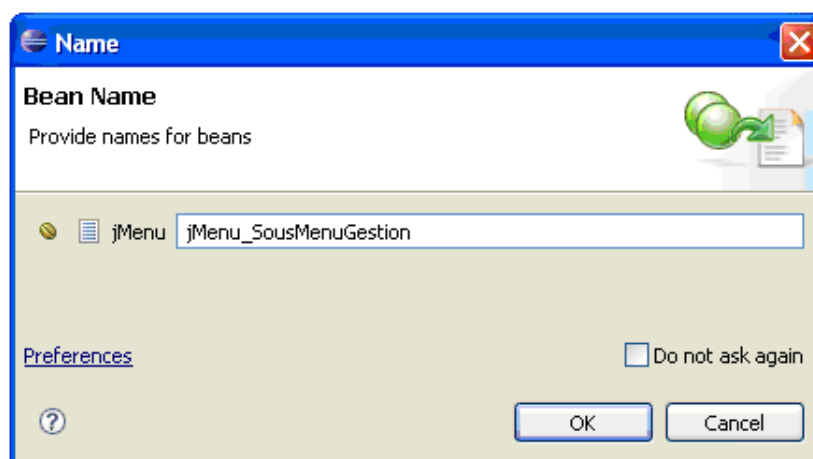
Pour faire simple, le menu **Fichier** comporte l'option **Quitter**, le menu **Edition** aucune et le menu **Clients** l'option **Liste**.

3. Ajout de sous-menus aux menus



Pour ajouter un sous-menu, il faut utiliser à nouveau le composant **JMenu**.

- Déplacez ce composant sur le menu **Clients** pour ajouter le sous-menu **Gestion** au menu **Clients**.



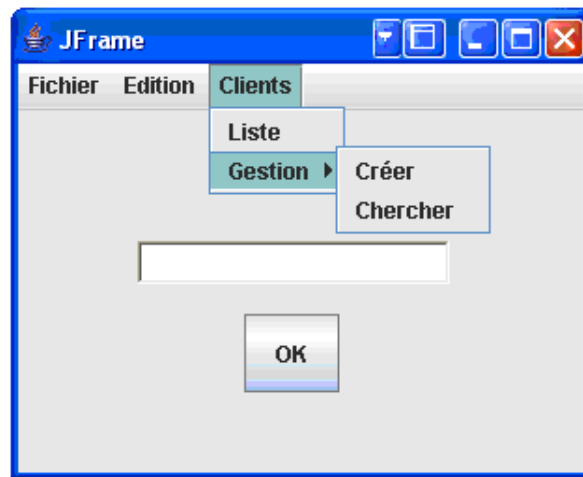
Le code du menu **Clients** s'est enrichi d'une ligne.

```
private JMenu getMenu_Clients() {
    ...
    jMenu_Clients.add(getJMenu_SousMenuGestion());
    ...
}
```

Et le code concernant le sous-menu **Gestion** a été créé.

```
private JMenu getMenu_SousMenuGestion() {
    ...
    jMenu_SousMenuGestion.setText("Gestion");
    ...
}
```

4. Ajout des options au sous-menu



Nous allons à nouveau utiliser le composant **JMenuItem** pour ajouter deux options au sous-menu **Gestion**, **Créer** et **Chercher**.



Il n'est pas possible de faire glisser le composant **JMenuItem** directement sur le sous-menu concerné.

- Déposez pour l'instant ce composant sur le menu **Clients**. Nous interviendrons au niveau du code pour le placer à l'endroit adéquat.

Code de l'option **Créer** :

```
private JMenuItem getJMenuItem_Créer() {
    if (jMenuItem_Créer == null) {
        jMenuItem_Créer = new JMenuItem();
        jMenuItem_Créer.setBounds(new Rectangle(237, 22, 29, 10));
        jMenuItem_Créer.setText("Créer");
    }
    return jMenuItem_Créer;
}
```

- Ajoutez l'option **Créer** dans le code du menu **Gestion**.

```
private JMenu getMenu_SousMenuGestion() {
    if (jMenu_SousMenuGestion == null) {
        jMenu_SousMenuGestion = new JMenu();
        jMenu_SousMenuGestion.setText("Gestion");
        jMenu_SousMenuGestion.add(getJMenuItem_Créer());
    }
}
```

```
    return jMenu_SousMenuGestion;
}
```

- Renouvelez l'opération pour l'option **Chercher**.

```
private JMenuItem getJMenuItem_Chercher() {
    if (jMenuItem_Chercher == null) {
        jMenuItem_Chercher = new JMenuItem();
        jMenuItem_Chercher.setText("Chercher");
    }
    return jMenuItem_Chercher;
}
private JMenu getJMenu_SousMenuGestion() {
    if (jMenu_SousMenuGestion == null) {
        jMenu_SousMenuGestion = new JMenu();
        jMenu_SousMenuGestion.setText("Gestion");
        jMenu_SousMenuGestion.add(getJMenuItem_Créer());
        jMenu_SousMenuGestion.add(getJMenuItem_Chercher());
    }
    return jMenu_SousMenuGestion;
}
```

N'oubliez pas de supprimer la référence de l'option Chercher dans le code du menu Clients.

 Visual Editor a inséré toutes les bibliothèques **Swing** et **Awt** nécessaires à la création de cette application graphique.

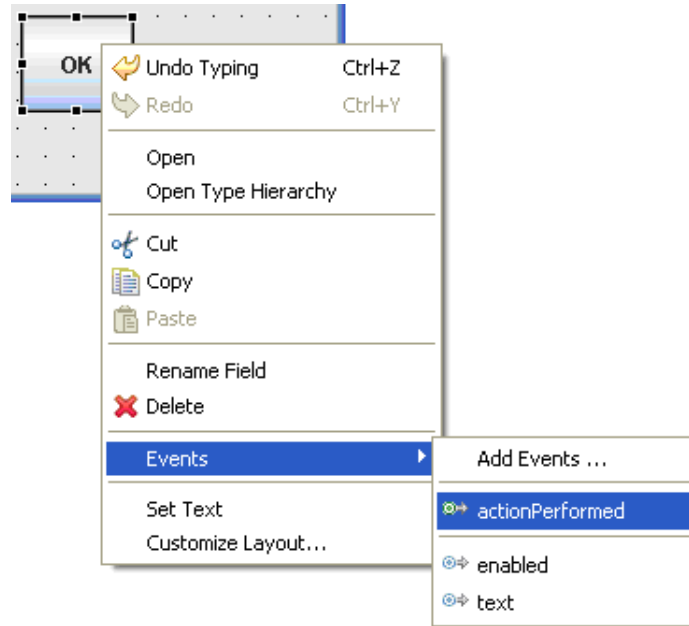
Il a également procédé à la déclaration de tous les composants utilisés.

```
private JPanel jContentPane = null;
...
private JButton jButton = null;
...
private JMenu jMenu_Clients = null;
...
```


Action d'un bouton

Nous voulons afficher un message de bienvenue avec le nom que l'utilisateur a saisi.

- Cliquez sur le bouton **OK**.
- Sélectionnez les options suivantes.




```
private JButton getJButton() {  
    if (jButton == null) {  
        jButton = new JButton();  
        jButton.setBounds(new Rectangle(119, 136, 51, 42));  
        jButton.setText("OK");  
        jButton.addActionListener(new java.awt.event.ActionListener() {  
            public void actionPerformed(java.awt.event.ActionEvent e) {  
                System.out.println("actionPerformed()");  
            }  
        });  
    }  
    return jButton;  
}
```

Il reste à remplacer la ligne **System.out.println("actionPerformed()")** par l'action voulue.

- Supprimez cette ligne et saisissez à la place le code suivant :

```
JOptionPane.showMessageDialog(null, "Bienvenue " + textField.getText());
```

Toutes ces notions sont détaillées dans le chapitre Développement.

 Pour atteindre les codes d'un élément de la fenêtre, sélectionnez cet élément puis appuyez simultanément sur les touches [Ctrl] et O (la lettre).

Composants

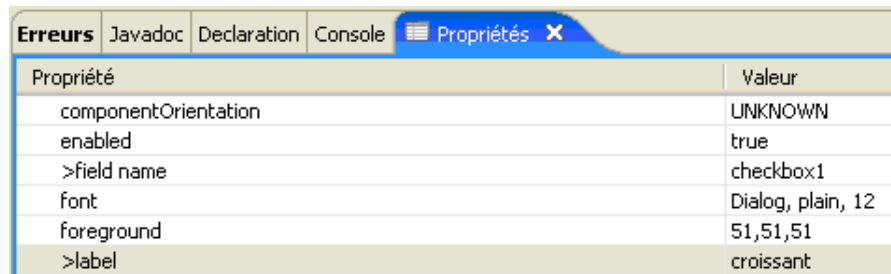
Tous les éléments graphiques sont fournis par les bibliothèques AWT (la plus ancienne) et Swing. Nous allons voir comment utiliser quelques autres composants classiques avec Visual Editor.

1. Case à cocher

- Créez une nouvelle fenêtre graphique en la nommant **FenCheckBox**.
- Cliquez sur le composant **jContentPane** et mettez la propriété **>layout** à **null**.

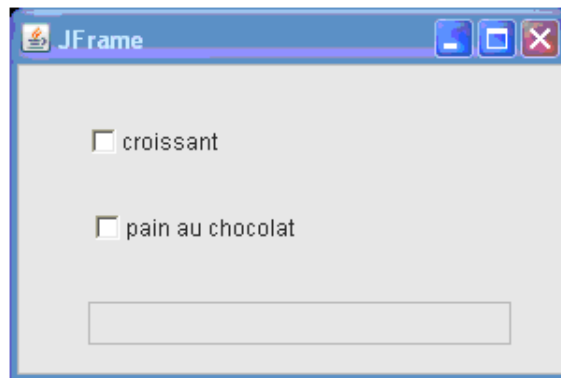
a. Avec AWT

- Déployez dans la palette l'onglet **AWT Controls** et déposez dans la fenêtre deux **CheckBox**.
- Nommez-les **checkboxbox1** et **checkboxbox2**.
- Utilisez l'onglet **Propriétés** et affectez les valeurs "croissant" et "pain au chocolat".

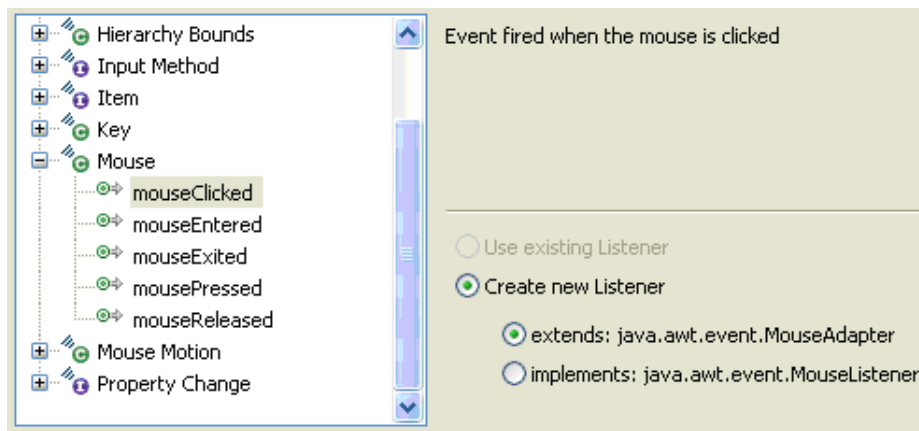


Propriété	Valeur
componentOrientation	UNKNOWN
enabled	true
>field name	checkboxbox1
font	Dialog, plain, 12
foreground	51,51,51
>label	croissant

- Déployez dans la palette l'onglet **Swing Components**, ajoutez un **JLabel** et supprimez le mot "JLabel" en face du champ **>text**.



- Testez la fenêtre en tant que **Java Bean**.
- Nous allons maintenant programmer les actions des cases à cocher.
- Effectuez un clic droit sur le **checkboxbox1** et sélectionnez les options suivantes.



- Supprimez l'action proposée par défaut et apportez les modifications suivantes :

```
private Checkbox getCheckbox1() {
    if (checkbox1 == null) {
        checkbox1 = new Checkbox();
        checkbox1.setBounds(new Rectangle(40, 30, 74, 23));
        checkbox1.setName("checkbox1");
        checkbox1.setLabel("croissant");
        checkbox1.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {boolean bSelectionne;
                bSelectionne = checkbox1.getState();
                if(bSelectionne == true){
                    jLabel.setText("Vous avez choisi un croissant");}
                else {
                    jLabel.setText("");
                }
            }
        });
    }
    return checkbox1;
}
```

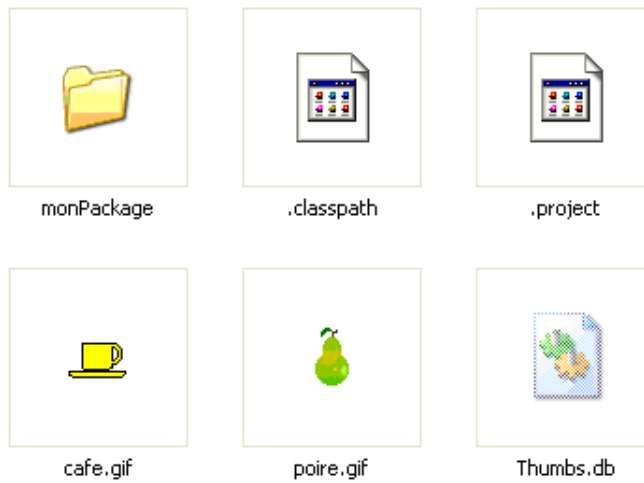
 Le Checkbox réagit au clic de souris. La méthode **getState()** permet de connaître l'état de la case : cochée ou non cochée.

La démarche est identique pour la deuxième case à cocher. Il est possible de concaténer les choix et de les afficher dans l'étiquette déjà présente ou tout simplement d'ajouter une deuxième étiquette.

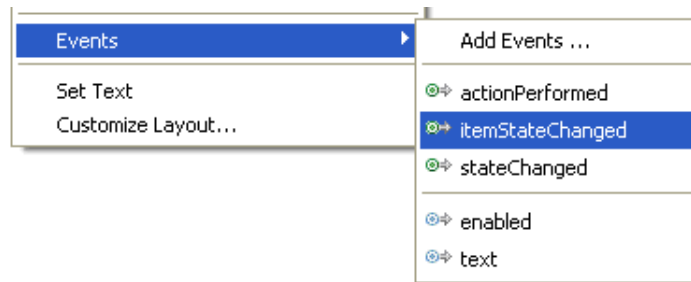
b. Avec Swing

La démarche avec les composants Swing est différente.

- Agrandissez la fenêtre et ajoutez un **JCheckBox** mais cette fois-ci de la bibliothèque Swing. Nommez-le **jCheckBoxSwing**.
- Ajoutez un nouveau **jLabel** et nommez-le **jLabelImage**. Pour apporter une variante aux tests, nous afficherons des images selon que la case est cochée ou décochée.
- À partir du disque dur, ajoutez au dossier du projet deux images. Par exemple :



- Effectuez un clic droit sur le **jCheckBoxSwing** et sélectionnez les options suivantes.



- Supprimez l'action proposée par défaut et apportez les modifications suivantes :

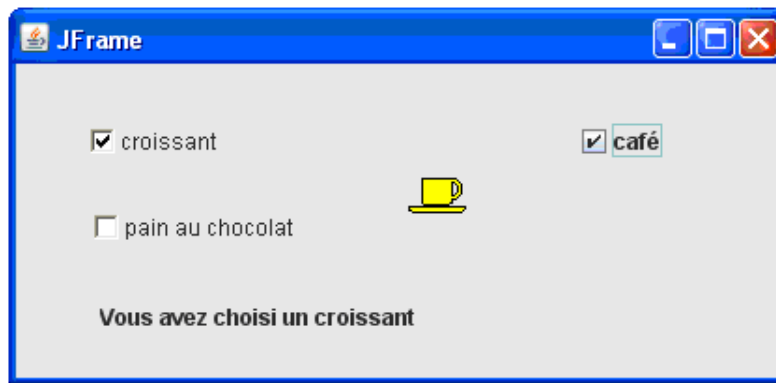
```
private JCheckBox getJCheckBoxSwing() {
    if (jCheckBoxSwing == null) {
        jCheckBoxSwing = new JCheckBox();
        jCheckBoxSwing.setBounds(new Rectangle(299, 31, 82, 21));
        jCheckBoxSwing.setText("café");

        jCheckBoxSwing.addItemListener(new java.awt.event.ItemListener() {
            public void itemStateChanged(java.awt.event.ItemEvent e) {
                System.out.println("itemStateChanged()");
                if(e.getStateChange() == ItemEvent.DESELECTED){
                    System.out.println("Je suis désélectionné");
                    jLabelImage.setIcon(new ImageIcon(""));
                }
                else{
                    System.out.println("Je suis sélectionné");
                    jLabelImage.setIcon(new ImageIcon("cafe.gif"));
                }
            }
        });
    }
    return jCheckBoxSwing;
}
```



Pour n'afficher aucune image, il suffit de passer en paramètre une chaîne vide.

- Testez la fenêtre.



2. Boutons radio

- Créez une nouvelle fenêtre graphique en la nommant **FenRadioButton**.
- Cliquez sur le composant **JContentPane** et mettez la propriété **>layout** à **null**.
- Déployez dans la palette l'onglet **Swing Components**, ajoutez trois **JRadioButton** et nommez-les par exemple bleu, blanc et rouge.
- Testez la fenêtre.

Il est possible d'avoir tous les boutons sélectionnés en même temps. Pour les rendre dépendants les uns des autres et ne permettre qu'à un seul bouton à la fois d'être sélectionné, il faut les regrouper.

- Ajoutez la propriété suivante qui permet de créer un objet groupe de boutons :

```
private ButtonGroup groupeBoutons = new ButtonGroup();
```

- Ajoutez la méthode suivante.

```
private void regroupeBoutons(){
    groupeBoutons.add(jRadioButton1);
    groupeBoutons.add(jRadioButton2);
    groupeBoutons.add(jRadioButton3);
}
```

- Modifiez le constructeur de la classe.

```
public FenRadioButton() {
    super();
    initialize();
    regroupeBoutons();
}
```

- Testez la fenêtre.

Nous allons programmer les actions.

- Pour chaque bouton, effectuez un clic droit et choisissez les options suivantes :




- Pour chaque bouton, précisez l'action.

```
private JRadioButton getJRadioButton1() {
    if (jRadioButton1 == null) {
        jRadioButton1 = new JRadioButton();
        jRadioButton1.setBounds(new Rectangle(32, 28, 75, 21));
        jRadioButton1.setText("Bleu");
        jRadioButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                jContentPane.setBackground(Color.blue);
            }
        });
    }
    return jRadioButton1;
}

private JRadioButton getJRadioButton2() {
    if (jRadioButton2 == null) {
        jRadioButton2 = new JRadioButton();
        jRadioButton2.setBounds(new Rectangle(32, 65, 75, 21));
        jRadioButton2.setText("Blanc");
        jRadioButton2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                jContentPane.setBackground(Color.white);
            }
        });
    }
    return jRadioButton2;
}

private JRadioButton getJRadioButton3() {
    if (jRadioButton3 == null) {
        jRadioButton3 = new JRadioButton();
        jRadioButton3.setBounds(new Rectangle(32, 103, 75, 21));
        jRadioButton3.setText("Rouge");
        jRadioButton3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                jContentPane.setBackground(Color.red);
            }
        });
    }
    return jRadioButton3;
}
```

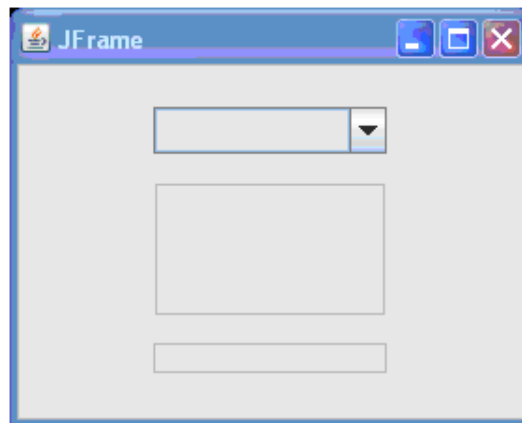
 Avec la méthode **actionPerformed()**, les boutons radio réagissent au clic de souris mais aussi à la touche [Espace] lorsqu'ils ont le focus.

- Testez la fenêtre.



3. Liste déroulante

- Créez une nouvelle fenêtre graphique en la nommant **FenListeDeroulante**.
- Cliquez sur le composant **jContentPane** et mettez la propriété **>layout** à **null**.
- Déployez dans la palette l'onglet **Swing Components**, ajoutez un **JComboBox**, nommez-le **listeDeroulante** et agrandissez-le.
- Ajoutez deux **JLabel** nommés **image** et **choix**.
- Supprimez le texte par défaut de la propriété **Text** pour les deux labels.



- Testez la fenêtre. Le JComboBox est bien sûr vide. Nous allons le remplir.
- Ajoutez la propriété suivante.

```
private String[] lesFruits = {"pomme", "banane", "poire"};
```



Un tableau de type chaîne est déclaré et initialisé avec des noms de fruits.

- Modifiez la ligne suivante dans la méthode **getListeDeroulante()**.

```
private JComboBox getListeDeroulante() {
    ...
    listeDeroulante = new JComboBox(lesFruits);
    ...
}
```



```

    return listeDeroulante;
}

```



Le tableau `lesFruits` est passé en paramètre au constructeur de `JComboBox`.

- Testez la fenêtre.

Nous voulons maintenant que la sélection s'affiche dans le **JLabel** nommé **choix**.

- Effectuez un clic droit sur le `JComboBox` et choisissez **Events - actionPerformed**.
- Modifiez le code généré.

```

private JComboBox getListeDeroulante() {
    if (listeDeroulante == null) {
        listeDeroulante = new JComboBox(lesFruits);
        listeDeroulante.setBounds(new Rectangle(73, 23, 125, 25));
        listeDeroulante.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String laSelection = (String) listeDeroulante.getSelectedItem();
                choix.setText(laSelection);
            }
        });
    }
    return listeDeroulante;
}

```



Le `JComboBox` est à l'écoute d'un événement extérieur (clic de souris ou barre espace) et effectue une action dès que cet événement survient. L'action proposée correspond à l'affichage de l'item sélectionné du `JComboBox` dans la deuxième étiquette nommée **choix**.

Nous voulons pour finir que l'image correspondant au fruit soit affichée dans la première étiquette nommée **image**. Ajoutez des images au projet par exemple :




- Modifiez à nouveau la méthode **getListeDeroulante()**.

```

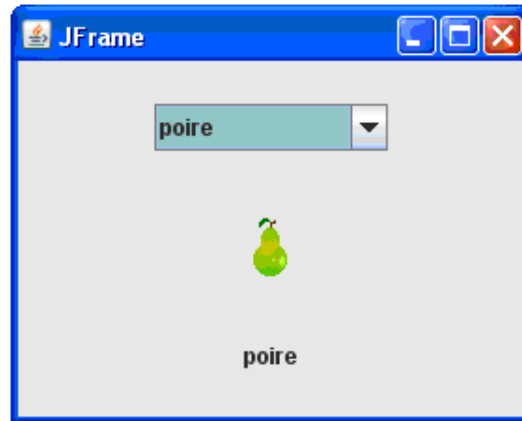
private JComboBox getListeDeroulante() {
    if (listeDeroulante == null) {
        listeDeroulante = new JComboBox(lesFruits);
        listeDeroulante.setBounds(new Rectangle(73, 23, 125, 25));
        listeDeroulante.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String laSelection = (String) listeDeroulante.getSelectedItem();
                choix.setText(laSelection);
                if (laSelection == "banane") {
                    ImageIcon imageFruit = new ImageIcon("banane.gif");
                    image.setIcon(imageFruit);
                }
                if (laSelection == "pomme") {
                    ImageIcon imageFruit = new ImageIcon("pomme.gif");
                    image.setIcon(imageFruit);
                }
                if (laSelection == "poire") {

```

```
        ImageIcon imageFruit = new ImageIcon("poire.gif");
        image.setIcon(imageFruit);
    }
});
}
return listeDeroulante;
}
```

 Le code proposé privilégie la simplicité pour faciliter la compréhension. Il est bien sûr possible d'effectuer une lecture des noms des images pour afficher la bonne image sans passer par une structure conditionnelle.


- Testez la fenêtre.

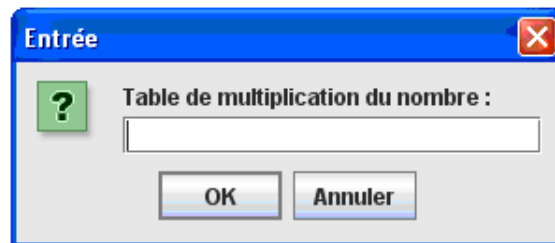


Débogueur

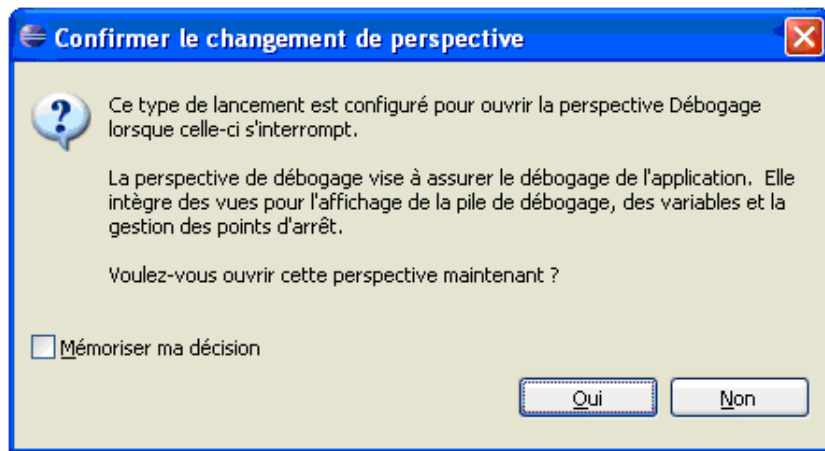
Le développement nécessite de nombreux tests. Pour contrôler la bonne exécution d'une partie d'un programme comportant des structures conditionnelles et itératives, l'utilisation d'un débogueur s'avère souvent indispensable. Nous allons voir comment utiliser celui d'Eclipse au travers de l'exemple classique d'un programme qui affiche les tables de multiplication. Nous vous proposons le code suivant :

```
1 package monPackage;
2
3 import javax.swing.JOptionPane;
4
5 public class TableMultiplication {
6     public static void main(String args[]) {
7         int vColonne, vLigne, Nombre = 0;
8         int tab [][] = new int[2][10];
9         Integer NB = Integer.valueOf(JOptionPane.showInputDialog("Table de " +
10             "multiplication du nombre : "));
11         Nombre = NB.intValue(); // le multiplicateur
12         for (vColonne=0; vColonne< tab.length; vColonne++) {
13             for (vLigne=0; vLigne< tab[vColonne].length; vLigne++){
14                 if (vColonne==0){
15                     tab[vColonne][vLigne]= vLigne+1;
16                 }
17                 else {
18                     tab[vColonne][vLigne] = Nombre *(vLigne+1);
19                 }
20             }
21         }
22         System.out.println("Table de " + Nombre);
23         System.out.println("-----");
24         for (vColonne=0; vColonne< tab.length; vColonne++) {
25             for (vLigne=0; vLigne< tab[vColonne].length; vLigne++){
26                 if (vColonne>0){
27                     tab[vColonne][vLigne] = NB*(vLigne+1);
28                     System.out.println(tab[vColonne-1][vLigne] + " x " + NB + " = "
29                         + tab[vColonne][vLigne]);
30                 }
31             }
32         }
33     }
34 }
```

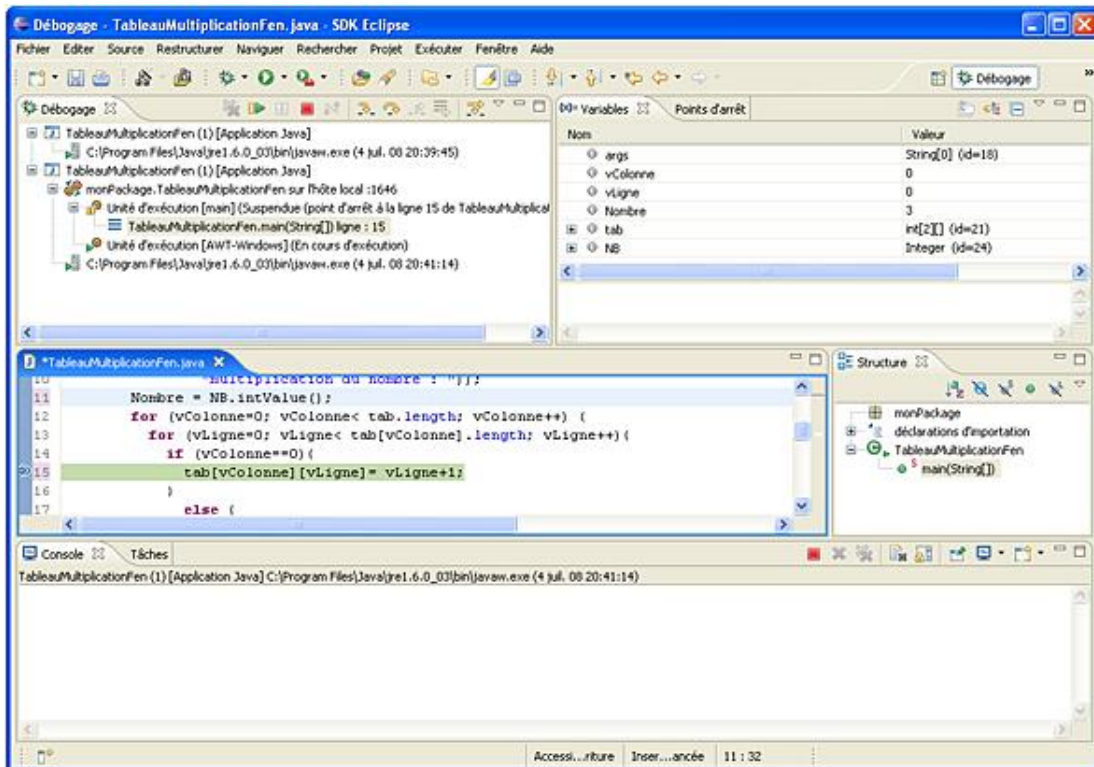
- Effectuez un clic droit dans la marge devant le numéro de ligne 15 et choisissez **Ajouter/Supprimer un point d'arrêt**.
- Testez l'application en cliquant sur l'icône de débogage dans la barre d'icônes ().



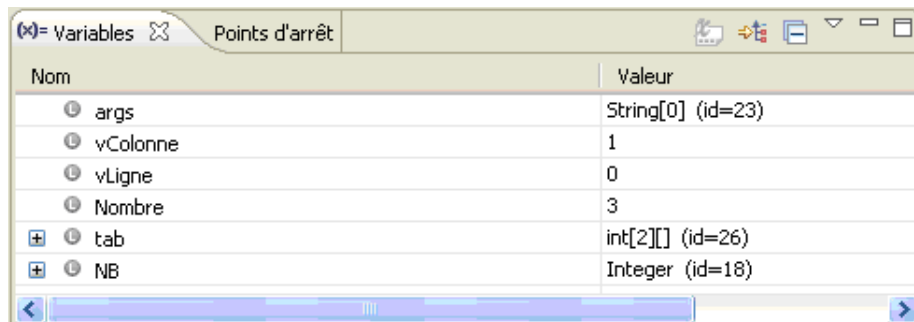
Pour une première utilisation, Eclipse affiche un message.




- Confirmez le changement de perspective.



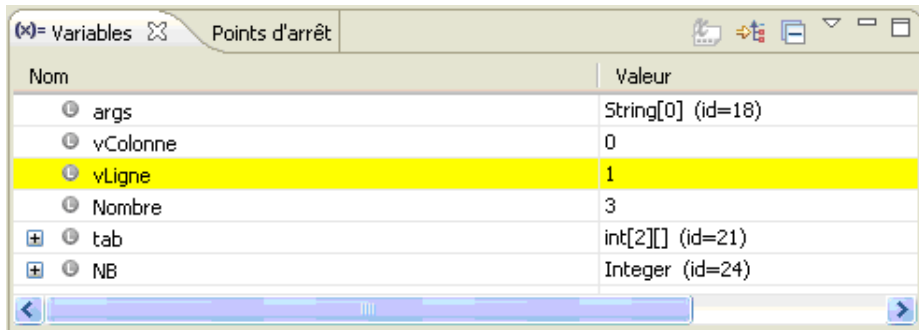
Les valeurs des variables sont affichées dans une fenêtre.



Le débogueur va nous permettre de vérifier l'exactitude des données en contrôlant pas à pas les valeurs des variables.

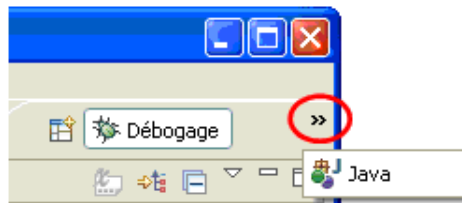
- Cliquez sur cette icône  .

Le point d'arrêt étant à la ligne 15, le parcours concerne la boucle for interne. A l'itération suivante, la variable vLigne doit contenir le nombre 1 et les variables vColonne et Nombre doivent demeurer inchangées.



Nom	Valeur
args	String[0] (id=18)
vColonne	0
vLigne	1
Nombre	3
tab	int[2][] (id=21)
NB	Integer (id=24)

- Pour revenir à la perspective Java, cliquez sur le bouton en haut à droite de la fenêtre.

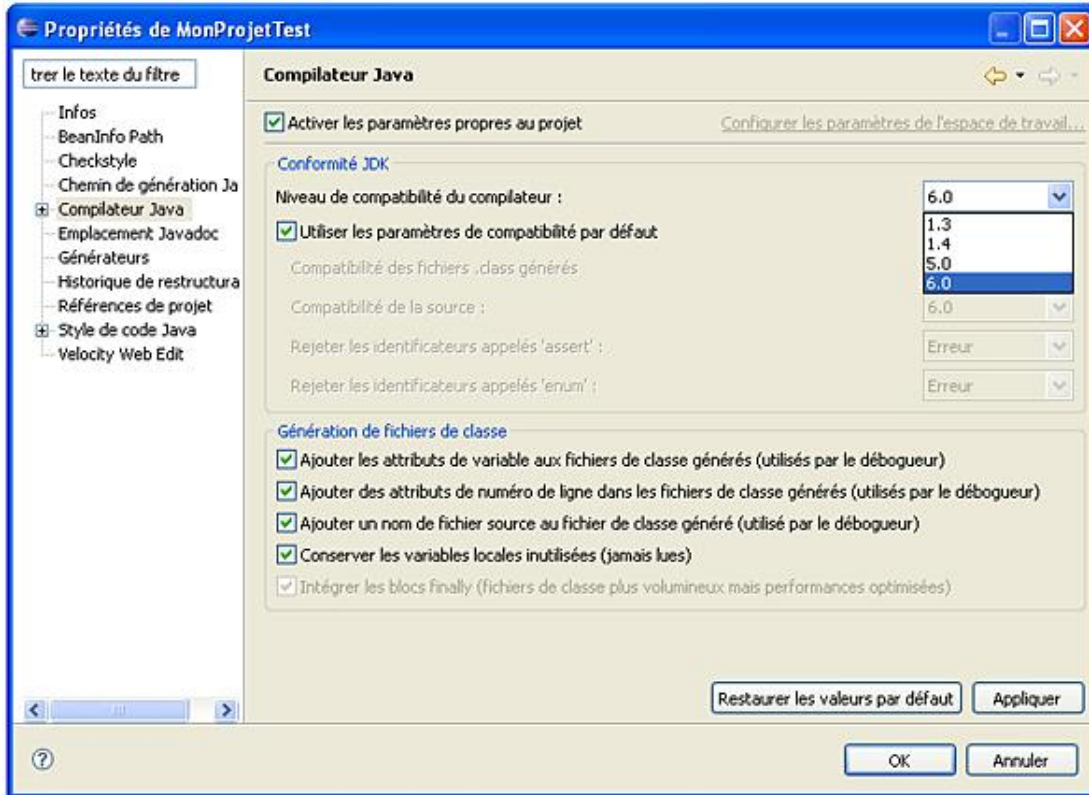


Propriétés du projet

Eclipse permet de personnaliser les propriétés de chaque projet.

- Dans la barre de menu, choisissez **Projet - Propriétés**.

Dans la fenêtre de propriétés, de nombreuses options de personnalisation sont disponibles. Cette fenêtre est particulièrement utile lorsqu'on désire assurer le niveau de compatibilité avec une version du JDK. Sélectionnez **Compilateur Java**. Cochez la case **Activer les paramètres propres au projet** et choisissez la version dans la liste déroulante.

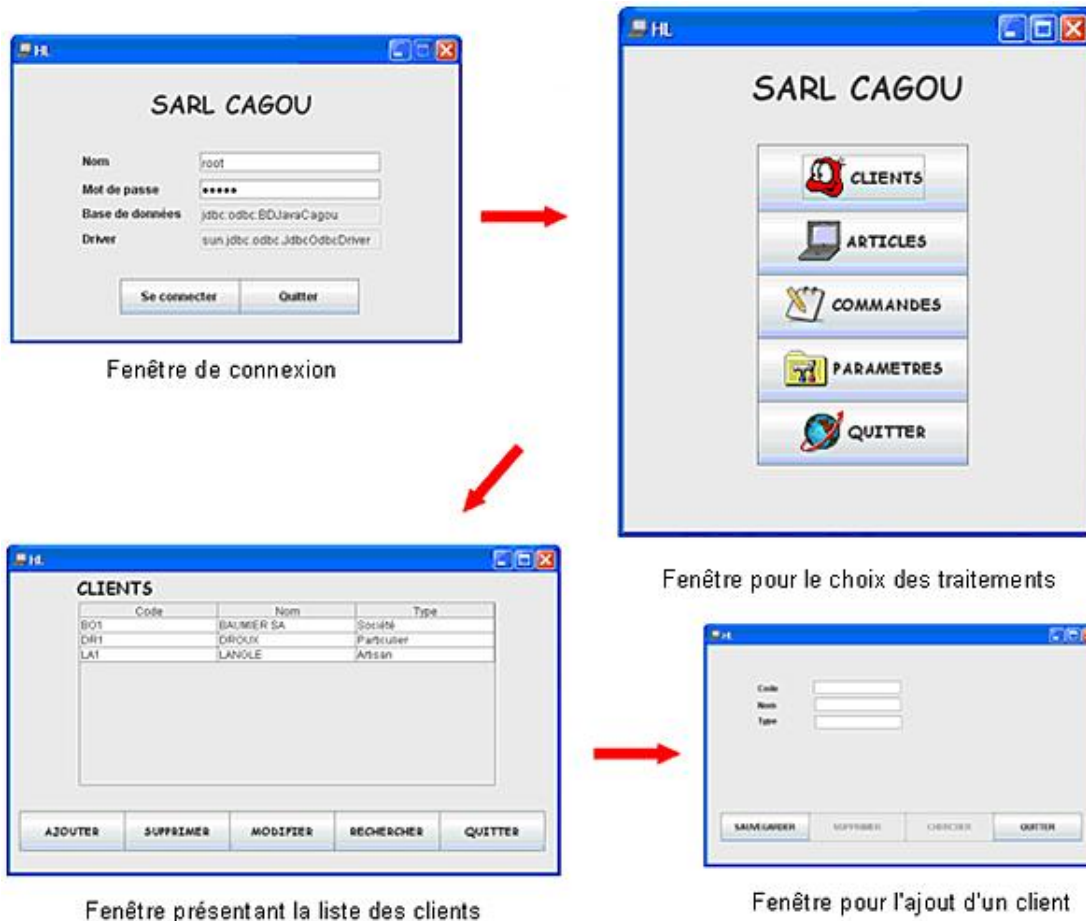


Application SA Cagou

L'application SA Cagou est une application classique de gestion. Elle prend en charge les commandes des clients. Dans le cadre de notre étude, l'objectif n'est pas de livrer une application clés en main. Il s'agit de comprendre et de maîtriser les principaux concepts objets à partir d'un cas concret. Il s'agit aussi de structurer au mieux le code en adoptant une méthode de développement.

Nous bornons donc le développement aux fonctionnalités suivantes :

- Gestion de l'accès.
- Gestion d'une page menu.
- Gestion des clients.



Voici les actions que l'utilisateur doit pouvoir réaliser :

- Saisie de son nom et de son mot de passe.
- Choix d'une option dans la page menu, ici l'option concernant les clients.
- Gestion des clients : consultation, ajout, suppression, modification, recherche.

En fonction des actions de l'utilisateur, l'application doit :

- signaler les erreurs ;
- refuser l'accès à la page menu tant que les paramètres d'accès sont incorrects ;

- afficher les fenêtres en fonction des traitements effectués ;
- afficher les données en mode fiche ou table ;
- demander une confirmation pour toute suppression.

Dans un souci d'ergonomie, les points suivants doivent être pris en compte :

- l'utilisateur doit pouvoir aussi bien utiliser le clavier que la souris pour effectuer un traitement ;
- le curseur de saisie doit être placé au bon endroit à l'ouverture d'une fenêtre ou suite à une action ;
- le double clic d'une ligne dans une table doit ouvrir une fenêtre présentant les données de cette ligne en mode fiche.

Persistence des données

La sauvegarde des données d'une application Java peut être réalisée de diverses manières. La plus simple est celle qui consiste à les sauvegarder dans des fichiers de type texte délimité mais c'est une solution obsolète et inadaptée aux applications de gestion. La sérialisation et le stockage dans des bases de données relationnelles sont des choix plus sérieux.

La sérialisation est une solution élégante qui permet d'obtenir la persistance des objets mais elle comporte plusieurs variantes présentant chacune des avantages et des inconvénients. Selon le type de sérialisation retenu, on est limité soit par la vitesse soit par le nombre de classes ou encore par leur complexité.

Les bases de données relationnelles sont incontournables dans le monde de l'informatique et plus particulièrement dans celui de la gestion. Elles reposent sur une technologie éprouvée depuis des décennies et ont su intégrer les données objets. Nous retenons donc pour notre projet Java cette option qui est par ailleurs celle actuellement la plus utilisée.

Parmi les nombreux SGBDR, MySQL est choisi pour ses qualités mais aussi parce qu'il est comme Eclipse un produit Open Source. Sa mise en œuvre et son utilisation avec Java ainsi que le mapping objet-relationnel sont décrits au chapitre Base de données MySQL.

4. Diagramme de communication

5. Diagramme de classes

6. Code

L'idée est finalement simple : pour modéliser (comprendre et représenter) un système complexe, il est nécessaire de recommencer plusieurs fois, en affinant son analyse par étape et en autorisant les retours sur les étapes précédentes. Cette démarche est appliquée au cycle de développement dans son ensemble. Une ou plusieurs itérations permettent ainsi une avancée significative nommée incrément. Le processus est réitéré à partir de chaque incrément.



La démarche itérative et incrémentale peut être aussi appliquée aux cycles de vie classique (en cascade, en V, ...).

Parvenu enfin au diagramme des classes, deux options sont possibles. Soit on commence de suite la production du code, soit on diffère celle-ci pour réaliser auparavant un regroupement des classes. Cet effort supplémentaire est guidé par la volonté de rationaliser le code afin de faciliter la maintenance corrective et évolutive.

Nous veillons ainsi à déterminer les classes en fonction de trois types :

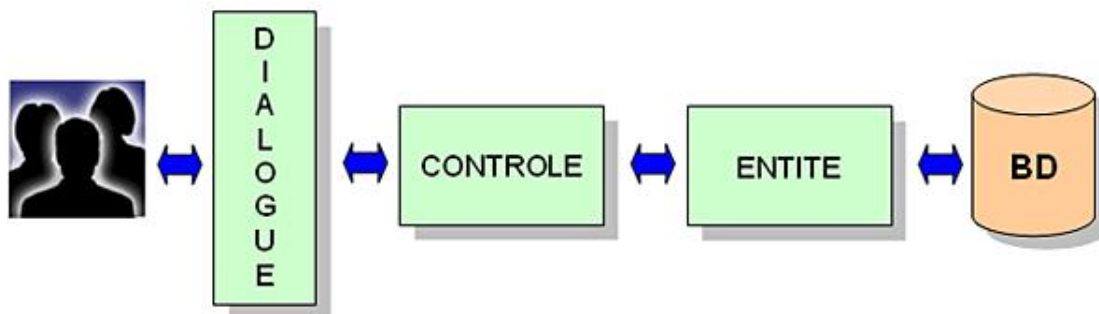
- dialogue
- contrôle
- entité

Le type dialogue regroupe les classes qui constitue l'interface homme-machine, IHM. Ce sont en majorité des classes graphiques issues des maquettes. Elles n'effectuent aucun traitement hormis ceux qui permettent des interactions avec l'utilisateur et sont responsables de l'affichage des données.

Les classes du type contrôle établissent la jonction entre les classes des deux autres types. Par exemple, la demande de création d'un client est prise en compte par une classe contrôle qui la transmet à la classe entité possédant cette méthode. On peut aussi leur confier le contrôle du respect des règles métiers. Les classes techniques sont également de ce type.

Le type entité correspond aux classes propres au domaine modélisé. Ces classes sont le plus souvent persistantes et sont implémentées pour notre projet dans une base de données MySQL. Elles disposent des méthodes dites CRUD (Create, Read, Update, Delete).

Nous aboutissons à une architecture n-tiers.




Ajout du plugin UML Eclipse

Il existe plusieurs plugins UML pour Eclipse, certains gratuits d'autres payants. Nous utilisons celui proposé par la société Omondo. Celle-ci propose une version complète et performante mais payante et une version gratuite qui permet tout de même de créer onze des diagrammes d'UML2.

Lors de l'écriture de cet ouvrage, les diagrammes ont été créés avec la version Free Edition 2.1.0 plus ancienne que celle proposée actuellement en téléchargement.

- Téléchargez la version Free Edition Eclipse 3.2 sur le site des Editions ENI à l'adresse suivante : <http://www.editions-eni.fr/>

 eclipseUML_E320_freeEdition_2.1.0.20061006.zip

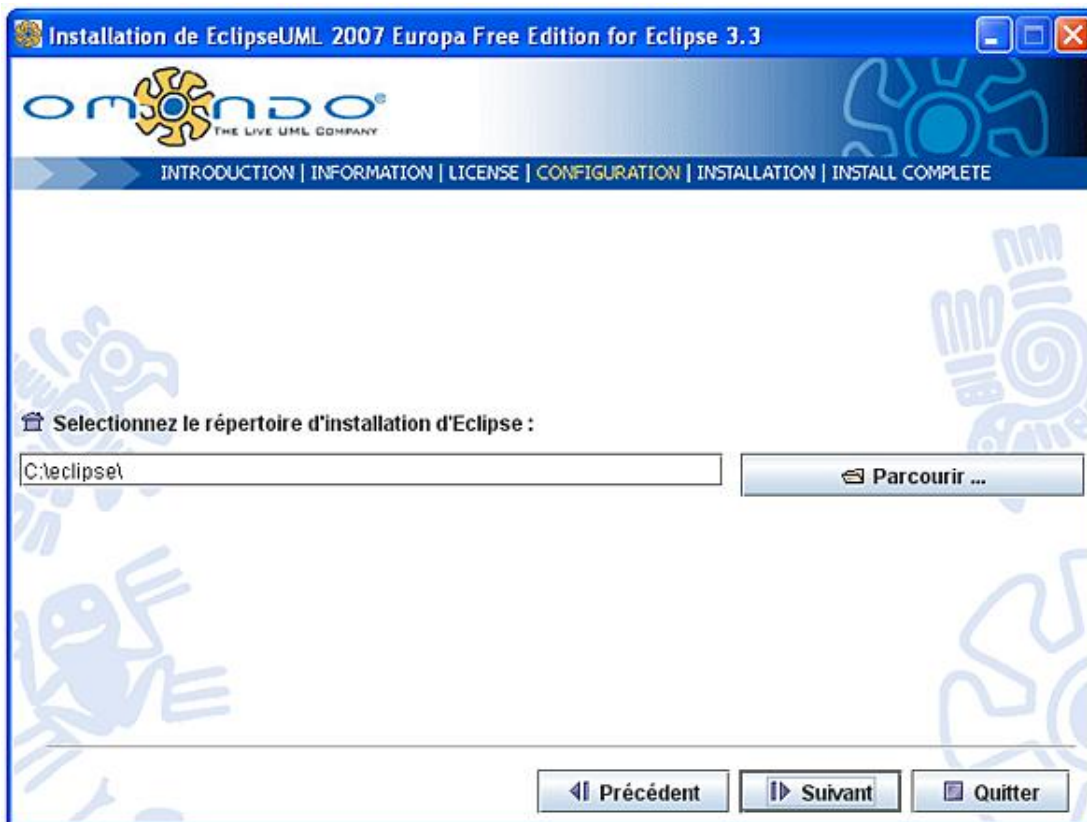
- Ou la dernière version si vous disposez d'Eclipse 3.3 à l'adresse suivante (celle-ci n'est pas à ce jour compatible avec Eclipse 3.2 : <http://www.omondo.com/>



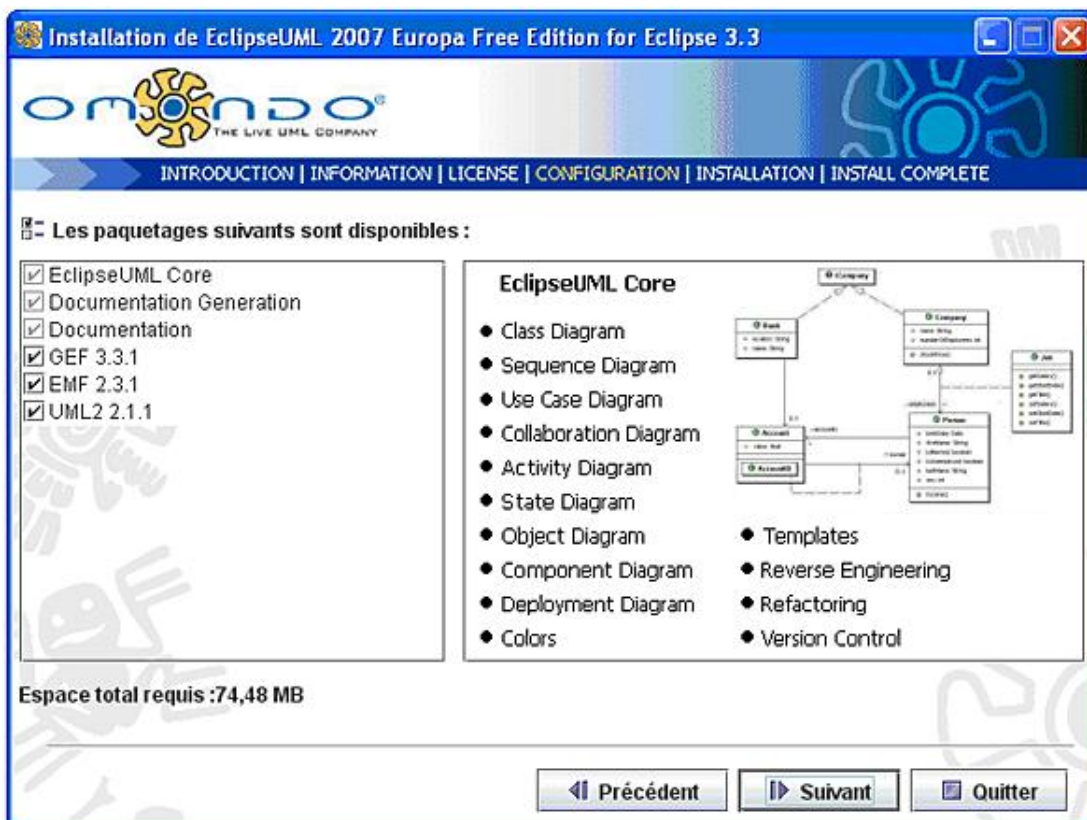
- Pour la version UML pour Eclipse 3.2, procédez à l'installation comme vue précédemment.
- Pour la nouvelle version pour Eclipse 3.3.X, procédez comme indiqué ci-après :



- Sélectionnez le dossier où se trouve Eclipse, sinon le plugin ne pourra pas être localisé et pris en compte. Exemple :



- Validez la sélection par défaut en cliquant sur le bouton **Suivant**.



Après l'installation vous pouvez modifier certains paramètres du plugin d'Eclipse selon vos préférences de travail.

- Dans le menu **Fichier**, choisissez le menu **Fenêtre** puis le sous-menu **Préférences....**

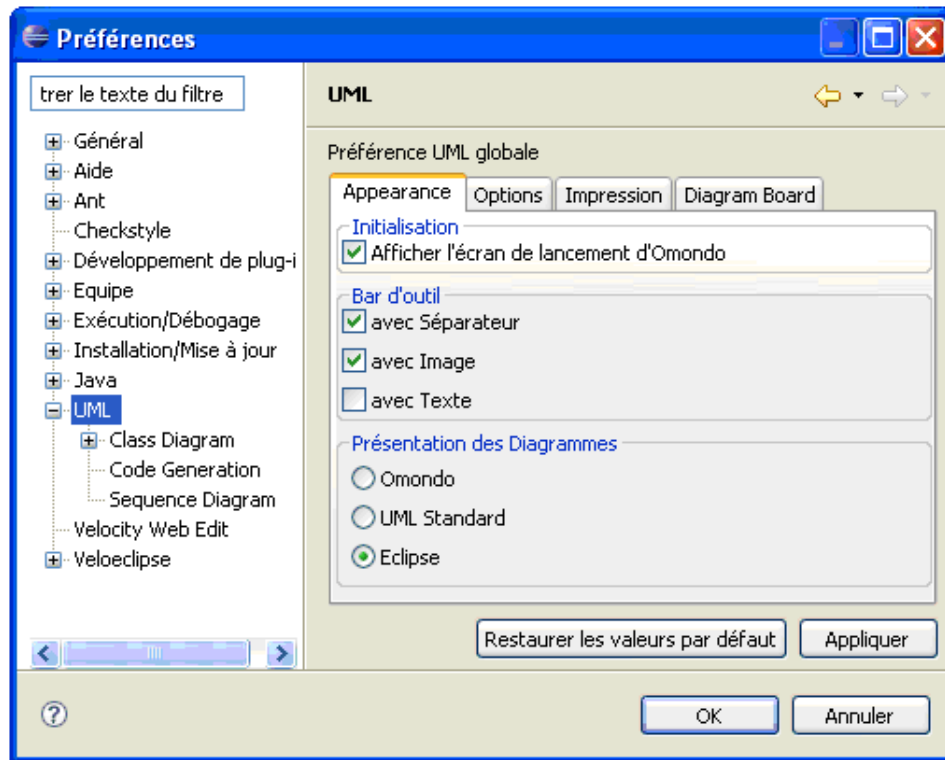


Diagramme de cas d'utilisation

Les diagrammes retenus pour notre projet sont présentés succinctement. Concernant leur étude approfondie, consultez les ouvrages traitant d'UML. Nous voyons ici leur mise en œuvre avec le plugin Free Edition d'Eclipse. Les diagrammes élaborés selon la démarche décrite précédemment vous sont proposés à titre indicatif.

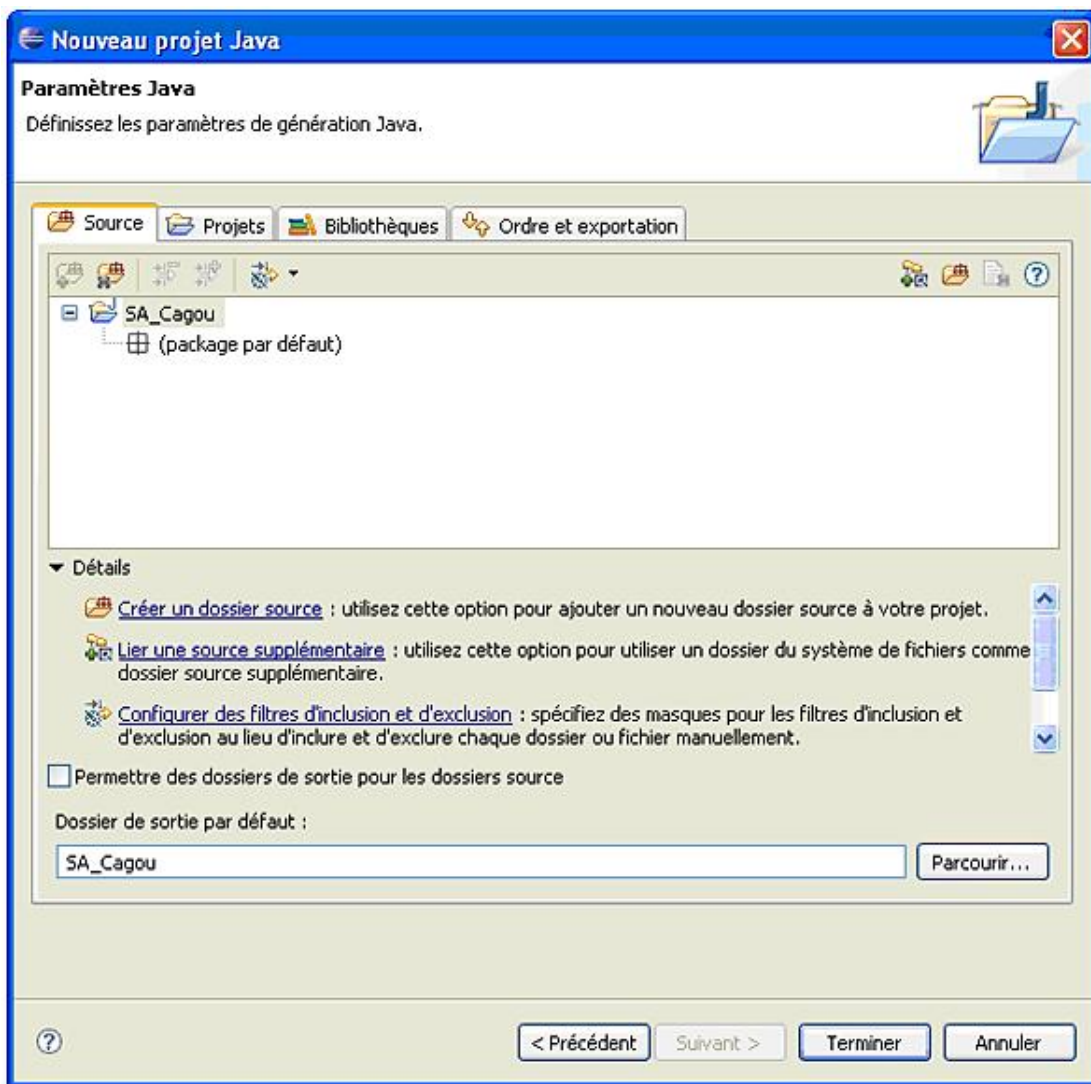
Le diagramme de cas d'utilisation comme souligné dans la section Démarche de ce chapitre, est à la fois le plus simple et souvent le moins utilisé. Nombre de développeurs lui attachent encore peu d'importance. Ceci est peut-être dû à l'héritage de Merise et de la plupart des premières méthodes objets (avant la fusion d'OMT, OOSE et Booch) pour lesquels il n'existe pas d'équivalent et dont l'élément principal, en ce qui concerne Merise, est le modèle conceptuel des données, modèle présentant des similitudes avec le diagrammes de classes.

Il s'agit avec le diagramme de cas d'utilisation de définir le comportement de la future application vis-à-vis de l'utilisateur ou de son environnement (autres systèmes informatiques, automates...). Il permet une compréhension commune entre le maître d'ouvrage et le maître d'œuvre, autrement dit entre le client, qu'il soit simple utilisateur ou expert, et le développeur. Les cas d'utilisation ne présentent que les interactions et les fonctionnalités attendues du système en cours de développement. Ils ne décrivent en rien les façons d'y parvenir. Durant le développement, ils servent de références permettant de vérifier que les besoins exprimés sont effectivement satisfaits.

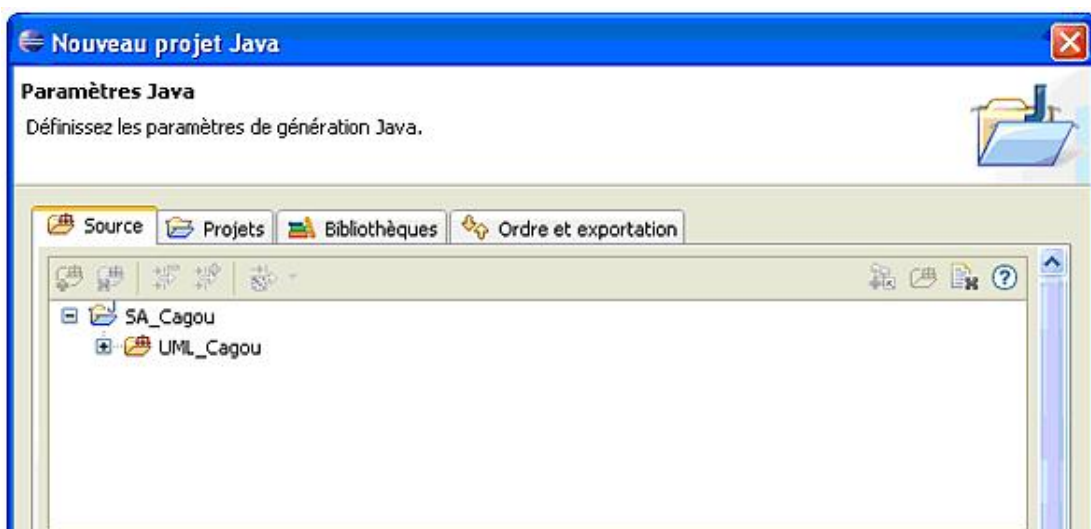
Nous allons commencer par créer le dossier du projet **SA_Cagou** et un sous-dossier **UML_Cagou** dans lequel nous rangerons nos classes.

- Dans le menu, choisissez **Fichier - Nouveau - Projet**.
- Ne saisissez rien. Cliquez sur le bouton **Suivant**.
- Nommez le projet **SA_Cagou** puis cliquez sur **Suivant**.

Vous parvenez à l'écran suivant :



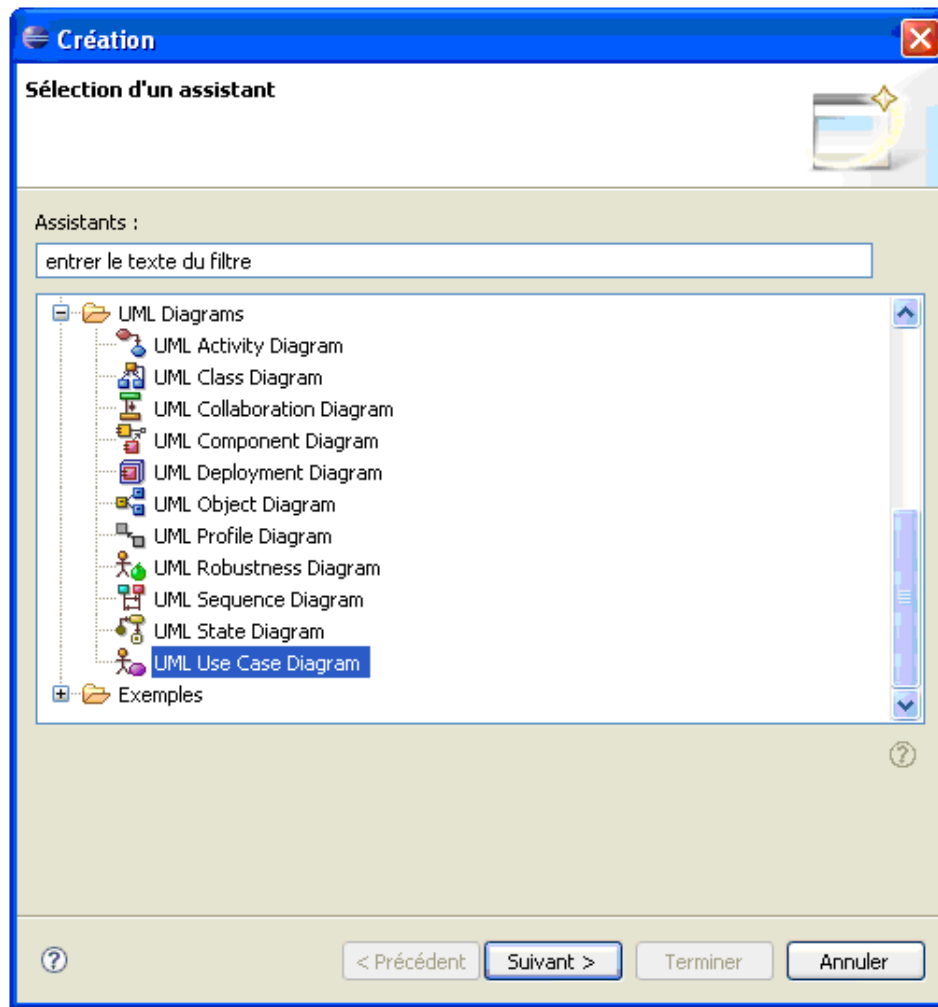
- Cliquez sur l'option **Créer un dossier source** puis saisissez **UML_Cagou** pour le nom du dossier puis cliquez sur **Terminer**.



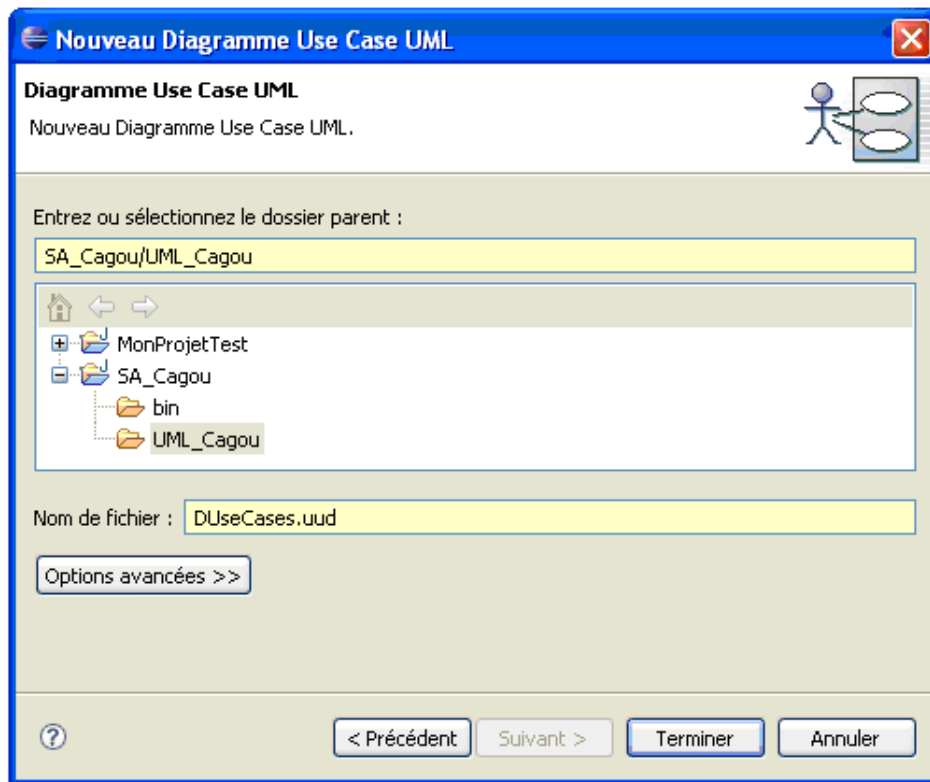
Nous allons maintenant créer le diagramme de cas d'utilisation.

- Sélectionnez le sous-dossier **UML_Cagou** et cliquez sur **Fichier - Nouveau - Projet - Autre**.

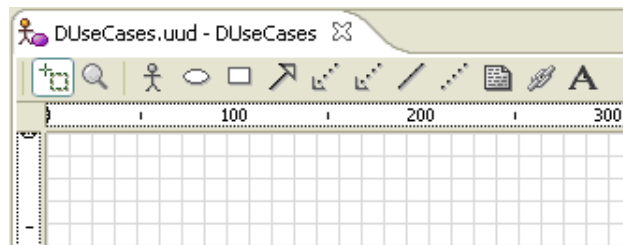
- Dans la fenêtre de création, déployez le dossier **UML Diagram**, sélectionnez **UML Use Case Diagram** puis cliquez sur le bouton **Suivant**.



- Sélectionnez le sous-dossier **UML_Cagou** et nommez le diagramme **DUseCases.uud** puis cliquez sur **Terminer**.



Le diagramme est généré. Bien entendu, il est vide. Pour commencer, nous allons retirer le quadrillage.



- Effectuez un clic droit sur le diagramme et choisissez **Propriétés**.
- Décochez l'option **Grid** dans l'onglet **Board properties**.
- Utilisez la barre d'outils pour bâtir le diagramme.

Le diagramme de cas d'utilisation proposé ci-après présente les interactions possibles de l'utilisateur avec notre système limité à la gestion des clients. Celui-ci doit pouvoir après identification, consulter, ajouter, supprimer, modifier et rechercher un client.

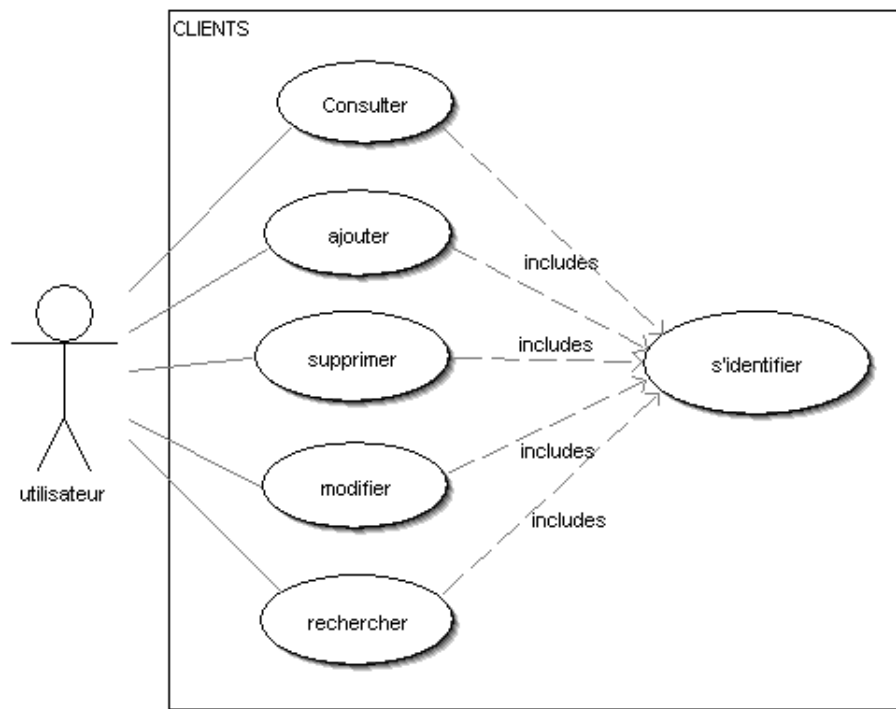


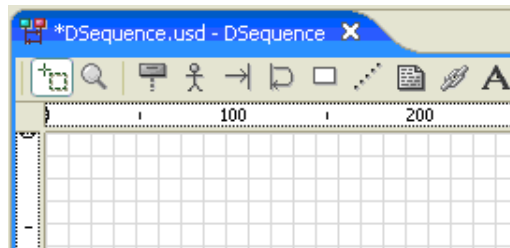
Diagramme de séquence

Le diagramme de séquence s'attache à modéliser l'aspect dynamique du système. Il peut être comparé à un storyboard mettant en évidence les interactions existantes entre des objets du système. L'accent est mis sur l'ordre chronologique des messages émis.

Pour chaque action de l'utilisateur définie dans le diagramme de cas d'utilisation, il convient d'établir un diagramme de séquence. Nous allons prendre pour exemple l'ajout d'un client.

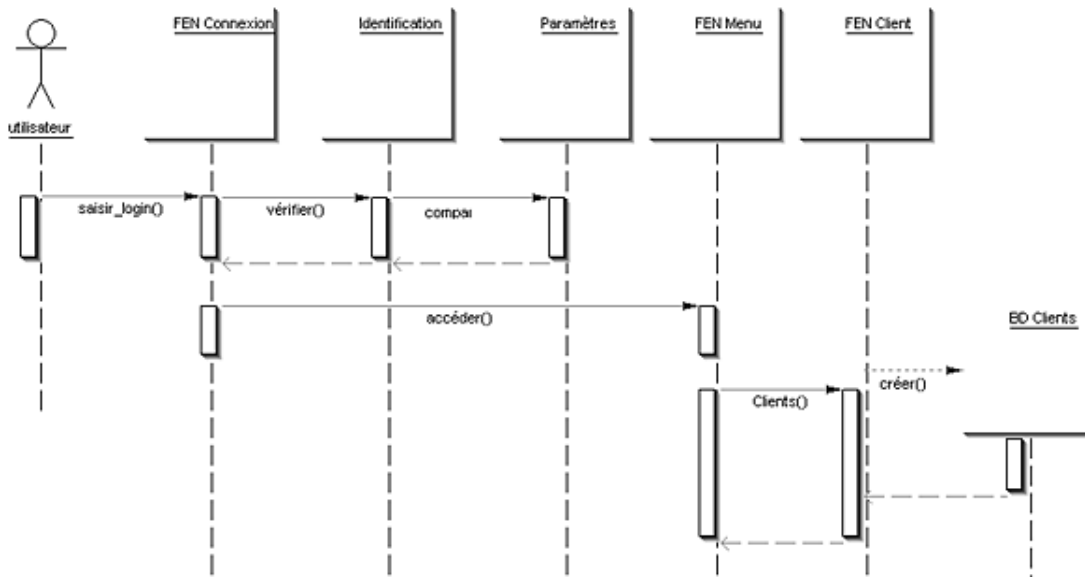
- Sélectionnez le projet **SA_Cagou** et cliquez sur **Fichier - Nouveau - Projet - Autre**.
- Dans la fenêtre de création, déployez le dossier **UML Diagram**, sélectionnez **UML Sequence Diagram** puis cliquez sur le bouton **Suivant**.
- Sélectionnez le sous-dossier **UML_Cagou** et nommez le diagramme **DSequence.usd** puis cliquez sur **Terminer**.

Le diagramme est généré avec une autre barre d'outils.



- Utilisez cette barre d'outils pour bâtir le diagramme.

Le premier diagramme proposé ci-après peut être utilisé si l'application est développée avec un langage non objet. L'accès à la base de données est direct.



Avec les langages orientés objet comme Java, toutes les tables de la base de données ont leur pendant en terme de classes. L'accès à la BD n'est donc pas direct car la demande par exemple d'ajout de l'utilisateur doit pour le moins transiter auparavant par une classe représentative de la table **Clients**.

Le second diagramme tient compte de ces remarques et comporte en outre un deuxième objet supplémentaire **Demandes** qui prend en compte la demande d'ajout dans notre exemple et la transmet à l'objet **Clients** pour traitement.

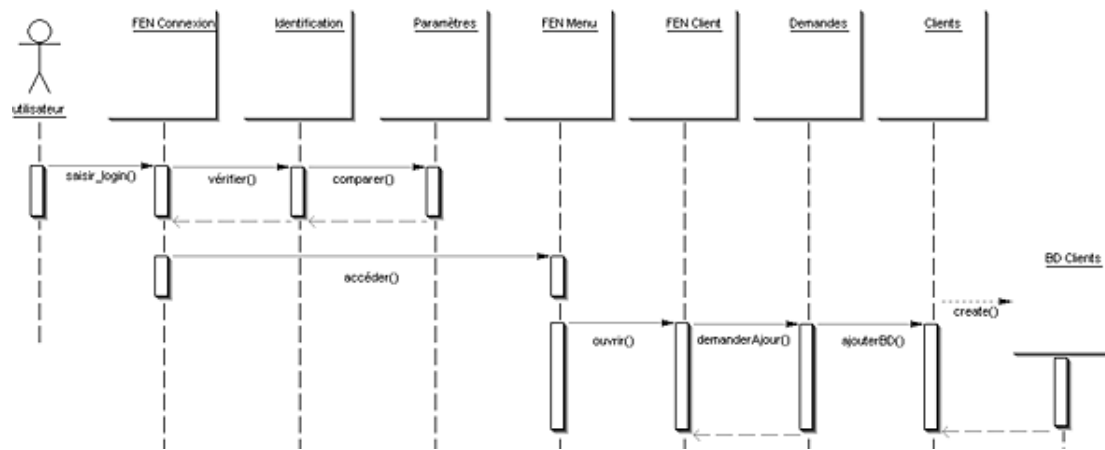


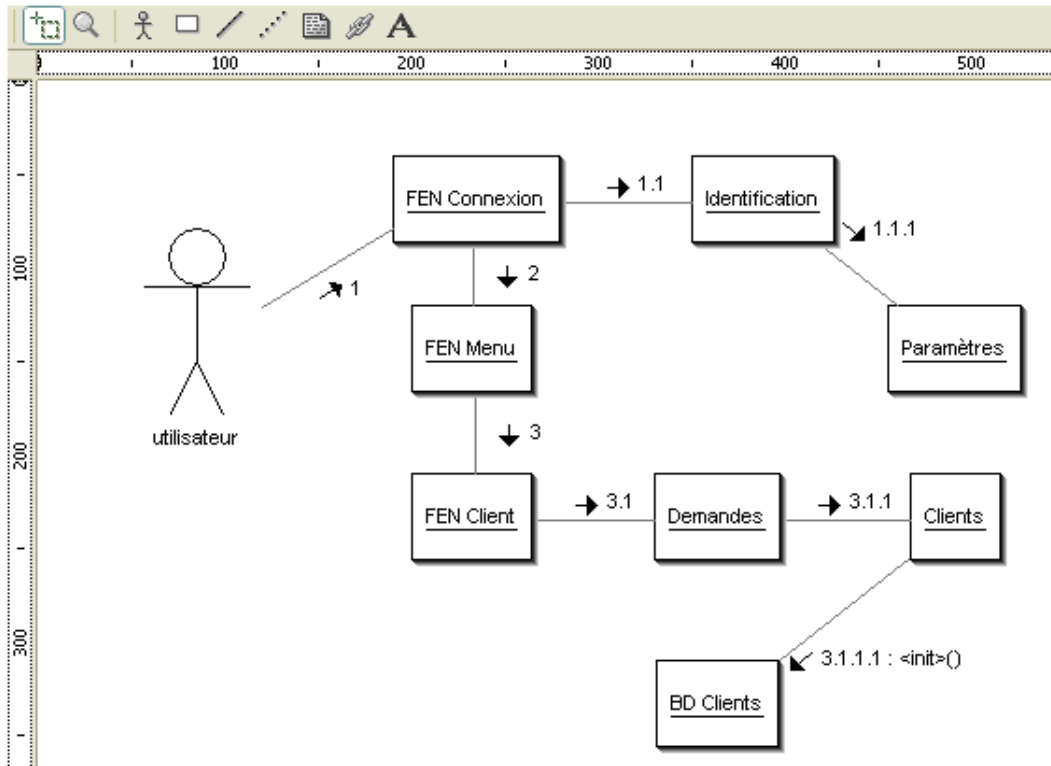
Diagramme de communication

Le diagramme de communication est équivalent du point de vue sémantique au diagramme de séquence. Il présente par contre l'organisation structurelle des objets tout en mettant comme précédemment en évidence les messages émis ou reçus.

Il est très facile de produire le diagramme de communication.

- Placez le curseur dans le diagramme de séquence en évitant les objets. Effectuez un clic droit et choisissez l'option **Generate collaboration diagram**.

Le diagramme est généré avec sa barre d'outils.



Les numéros correspondent aux méthodes du diagramme de séquence. Par exemple, le n° 1.1 correspond à **vérifier()**.

- Dans le diagramme de séquence, effectuez un double clic sur le mot **vérifier()** ou sur sa flèche.

Message

Propriété du message

Label:

Opération

Nombre de la Séquence

☐ Création

☐ Destruction

☒ Flèche retour

Synchronisation

☒ Atomic delivery


 Eclipse a conservé l'appellation d'UML 1 pour désigner le diagramme de communication. Avec la version Free Edition, il n'est pas possible de générer le diagramme de séquence à partir du diagramme de communication.

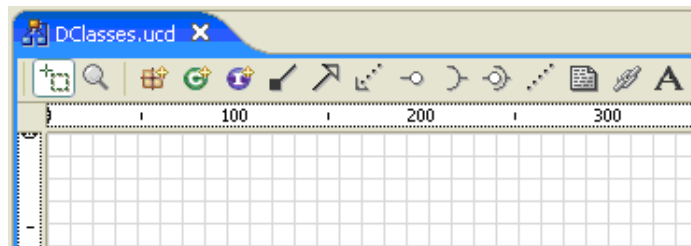
Diagramme de classes

Le diagramme de classes est utilisé pour modéliser l'aspect statique d'un système. Il met en évidence les classes et les relations qu'elles entretiennent entre elles : dépendances, association, généralisation, etc. Selon l'importance du système à modéliser, des diagrammes de classes intermédiaires peuvent être utilisés notamment les diagrammes des classes participantes qui décrivent, pour chaque cas d'utilisation, les trois principales classes d'analyse et leurs relations (cf. dans ce chapitre, le titre Démarche).

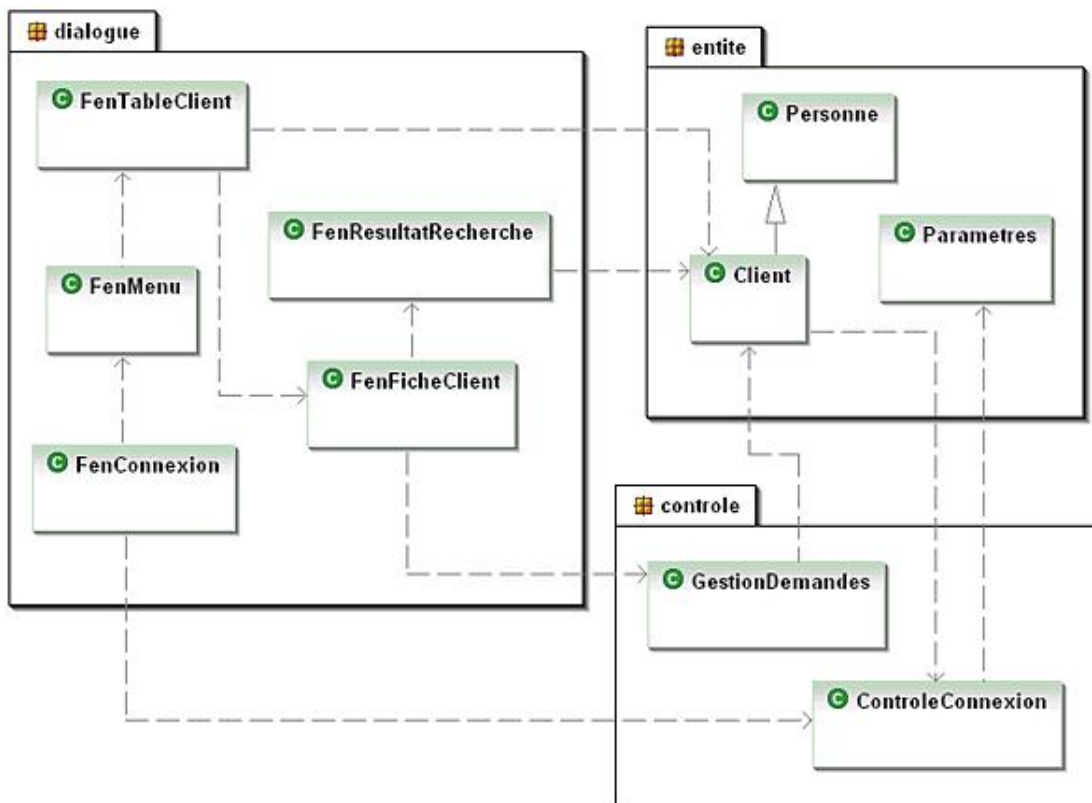
Selon la démarche retenue, nous pouvons (enfin) à ce stade produire le diagramme de classes. Il doit comporter les classes mises en évidence par les objets présents dans les diagrammes de séquence et de communication ainsi que celles éventuellement manquantes, nécessaires au fonctionnement des maquettes (cf. titre Démarche).

- Sélectionnez le projet **SA_Cagou** et cliquez sur **Fichier - Nouveau - Projet - Autre**.
- Dans la fenêtre de création, déployez le dossier **UML Diagram**, sélectionnez **UML Class Diagram** puis cliquez sur le bouton **Suivant**.
- Sélectionnez le dossier **UML_Cagou**, nommez le diagramme **DClasses.ucd** puis cliquez sur **Terminer**.

Le diagramme est généré avec sa barre d'outils.



Le diagramme des classes proposé découle des diagrammes de séquence et de communication **Ajout d'un client** et de ceux (non reportés ici) représentant les autres actions de l'utilisateur recensées dans le diagramme de cas d'utilisation. S'ajoute également une classe graphique supplémentaire **FenResultatRecherche** issue des maquettes qui permet d'afficher le jeu d'enregistrements d'une requête SQL dans une fenêtre.




La classe **FenConnexion** possède la méthode **main()**. C'est donc la première classe qui est exécutée. Elle appelle

ensuite la classe **ContrôleConnexion** qui a besoin de la classe **Parametres**. Si les paramètres de connexion sont corrects, la classe **FenMenu** est chargée. Celle-ci est une fenêtre qui propose différents choix de traitements. Rappelons que nous avons limité l'étude à la gestion des clients.

Nous pouvons constater que les classes de type vue, **FenTableClient** et **FenResultatRecherche** ont un accès direct à la classe **Client**. Il est d'usage d'accorder cette autorisation dès lors qu'il s'agit d'une simple lecture de la base de données.

Il est bien entendu possible d'ajouter à la classe **GestionDemandes**, une méthode de consultation mais cela n'aurait que peu d'intérêt et alourdirait le code. Par contre, toutes les demandes d'opérations pouvant modifier l'état de la base de données doivent transiter par la classe **GestionDemandes**. Les demandes sont ensuite transmises à la classe **Client** qui possède toutes les méthodes de type **CRUD** (Create, Read, Update, Delete).

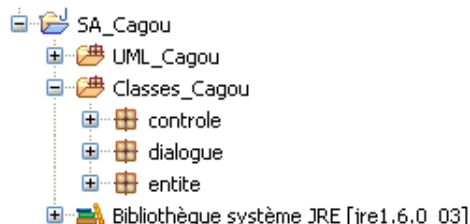
Enfin, nous observons une relation d'héritage entre les classes **Personne** et **Client**.

 Ce diagramme basé sur les trois types de classes d'analyse présentés précédemment propose un découpage semblable à celui proposé par le modèle **MVC**, *Modèle Vue Contrôleur*. MVC, créé en 1980 par Xerox PARC pour le langage Smalltalk, est un patron de conception logicielle ou **design pattern** qui introduit en outre le concept de synchronisation. L'idée est de mettre à jour automatiquement toutes les vues présentant les mêmes données en fonction des modifications apportées à la base de données et réciproquement. Cette mise à jour peut ne concerner que des vues suite à des actions de l'utilisateur. C'est ce qui différencie fondamentalement MVC de l'approche que nous avons mise en œuvre, celle-ci visant essentiellement à la séparation des données de l'interface homme-machine via une couche chargée de la transmission des demandes et des résultats.

MVC ne présente toutefois pas que des avantages. Pour que la synchronisation puisse être réalisée, il faut créer de nouvelles interfaces et classes de type écouteur et gestionnaire d'événements puis les ajouter aux modèles afin qu'ils puissent écouter et notifier les changements qui les concernent. Les contrôleurs doivent être écrits de sorte qu'ils puissent être informés des changements afin de procéder à la synchronisation et à la mise à jour des vues ou des modèles. Cette architecture apporte donc un niveau de complexité élevé impliquant un important travail de conception et une augmentation importante du code. Par ailleurs, la synchronisation peut être tributaire de nombreuses conditions ce qui alourdit d'autant sa mise en œuvre.

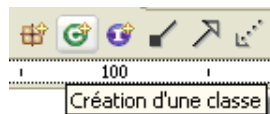
Voyons ensemble comment bâtir le diagramme de classes. Commençons par ajouter au projet un autre sous-dossier **Classes_Cagou** qui contiendra trois paquetages portant les noms de nos classes d'analyse : **dialogue**, **contrôle** et **entite** (ne mettez pas d'accent).

- Sélectionnez le projet **SA_Cagou** et cliquez sur **Fichier - Nouveau - Projet - Dossier source**. Nommez-le **Classes_Cagou**.
- Ajoutez les trois paquetages.



Nous allons maintenant créer nos classes en commençant par les classes de type entité. Nous les rangerons dans le paquetage **entite**.

- Cliquez sur cette icône.



- Sélectionnez le dossier source et le paquetage. Saisissez **Personne** pour le nom de la classe. Laissez les paramètres par défaut et cliquez sur le bouton **Suivant**.

Nouvelle classe Java

Classe Java
Créez une classe Java

Dossier source : SA_Cagou/Classes_Cagou Parcourir...

Package : modele Parcourir...

☐ Type englobant : Parcourir...

Nom :

Modificateurs : ☒ public ☐ Valeur par défaut ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclasse : Parcourir...

Interfaces : Ajouter... Supprimer

Quels raccords de méthode voulez-vous créer ?
☐ public static void main(String[] args)
☐ Constructeurs de la superclasse
☒ Méthodes abstraites héritées

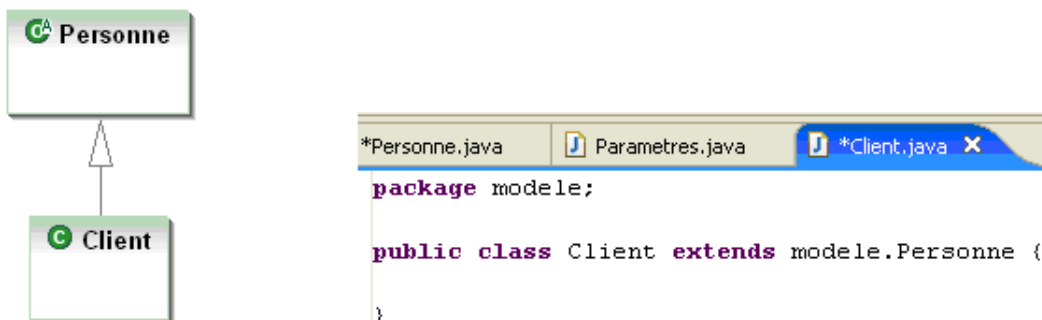
Voulez-vous ajouter des commentaires conformément à la configuration spécifiée dans les [propriétés](#) du projet sélectionné ?
☐ Générer les commentaires

? < Précédent Suivant > Terminer Annuler

De nombreux paramètres sont disponibles et leur choix allège l'écriture du code. Nous prenons l'option dans le cadre de cet apprentissage de ne rien cocher ce qui vous permettra de coder vous-mêmes en suivant les explications données au chapitre Développement.

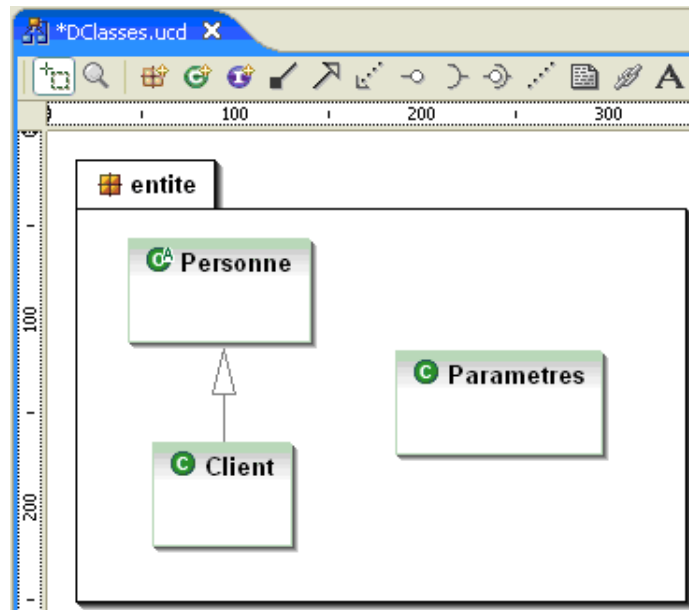
La classe apparaît dans l'explorateur de paquetages et le code correspondant à cette classe est généré. Effectuez au besoin un double clic sur la classe pour faire apparaître le code.

- Créez les deux autres classes entités, **Client** et **Parametres**.
- Matérialisez la relation d'héritage dans le diagramme de classes. Le code est automatiquement mis à jour.



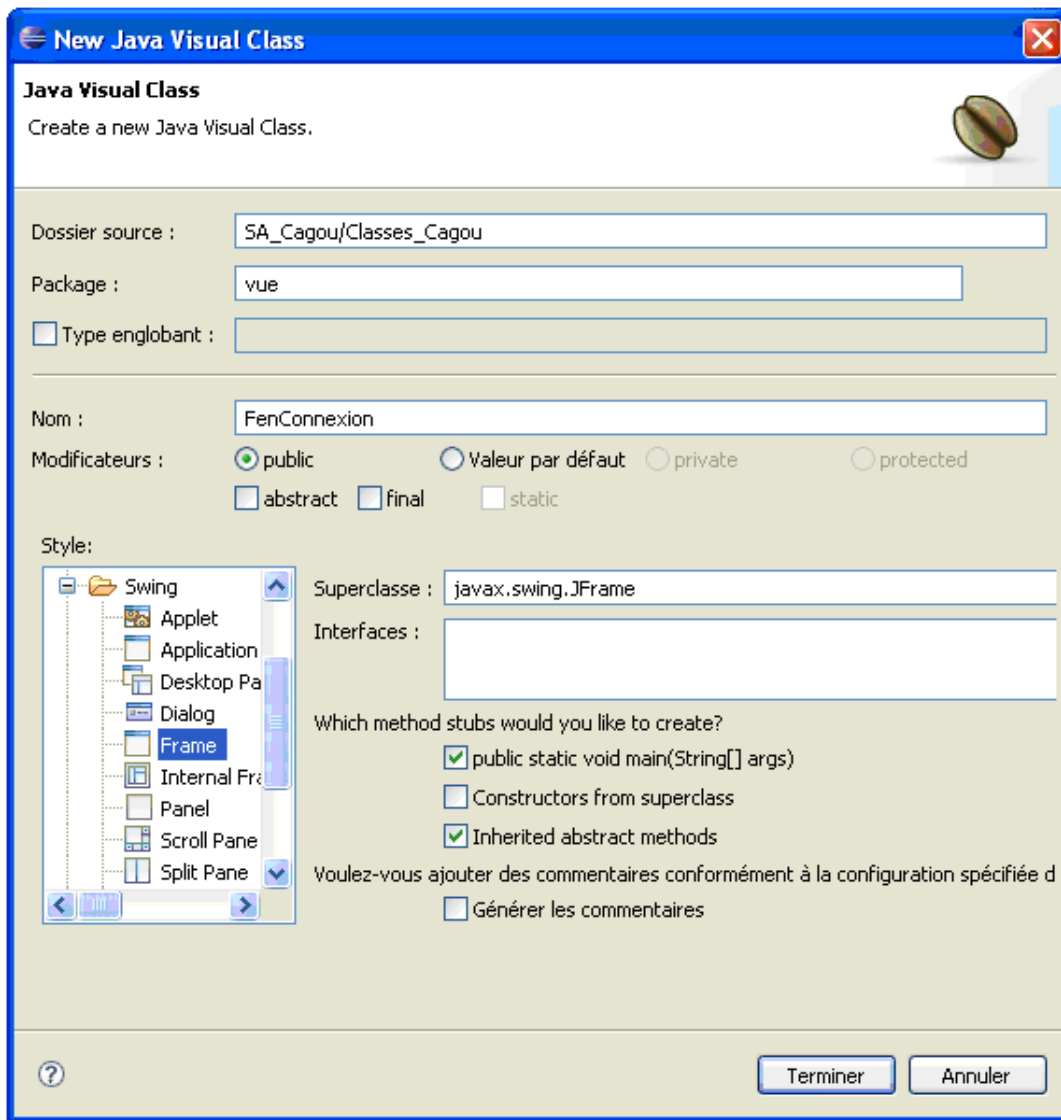
Nous allons maintenant regrouper ces classes dans leur paquetage.

- Cliquez sur le paquetage **entite** dans l'explorateur et déposez-le dans le diagramme de classes. Déplacez ensuite les classes dans ce paquetage.

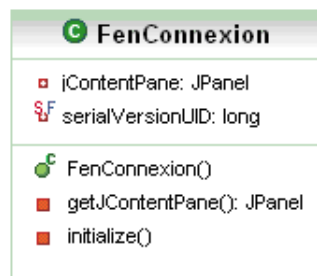


Pour les classes graphiques, nous allons procéder différemment afin de bénéficier des avantages apportés par le plugin Visual Editor (cf. chapitre Environnement de développement).

- Faites glisser le paquetage **dialogue** de l'explorateur de paquetages vers le diagramme. Agrandissez-le pour qu'il puisse recevoir la représentation graphique des cinq classes.
- Dans le menu, choisissez **Fichier - Nouveau - Projet - Visual Class**.
- Saisissez **FenConnexion** puis validez selon les paramètres ci-après. Cette fenêtre comporte la méthode principale **main()**.



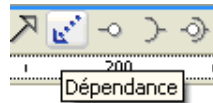
- Renouvelez l'opération pour les quatre autres fenêtres.
- Sélectionnez dans l'explorateur de paquets, la classe **FenConnexion**.



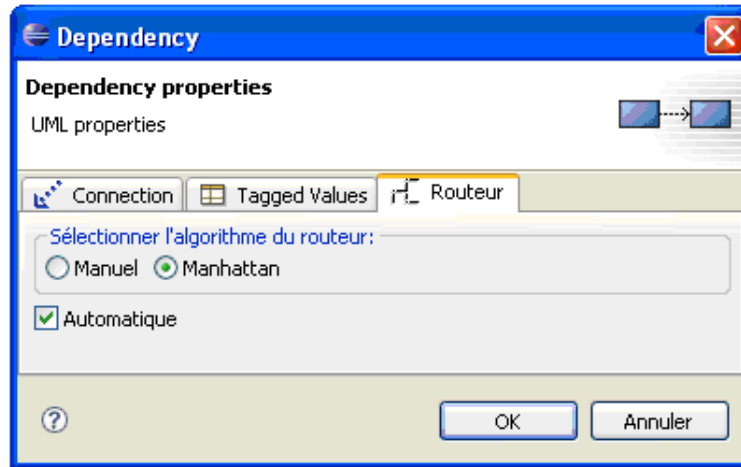
La classe graphique apparaît avec ses propriétés, le constructeur et ses méthodes. Vous allez les masquer pour gagner de la place.


- Effectuez un clic droit sur le titre de la classe et choisissez **Show/Hide Compartment** puis **Name Compartment only**.
- Renouvelez l'opération pour les quatre autres fenêtres.
- Ajoutez maintenant le paquetage **contrôle** et ses classes au diagramme en vous basant sur les opérations réalisées pour le paquetage **entité**.

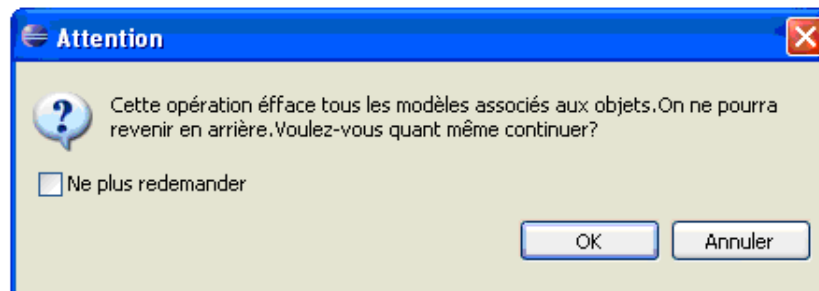
- Établissez les dépendances entre les classes et les paquetsages.



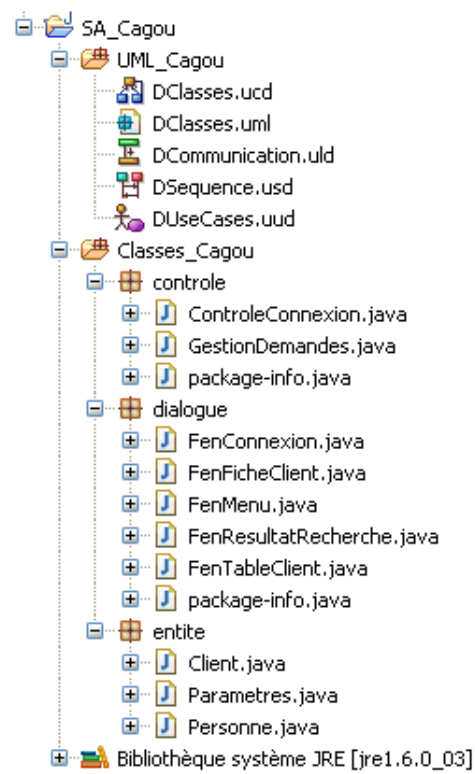
- Pour modifier le type de connexion, double cliquez sur les flèches de dépendance.



 Vous pouvez masquer ou supprimer une classe du diagramme. Attention, la suppression d'une classe du diagramme la supprime également du projet. Il est impossible d'annuler l'opération. Le message proposé par Eclipse n'est pas très explicite à ce sujet.



Au terme de l'analyse, voici la structure obtenue sous Eclipse :



Installation et configuration du serveur WAMP

Pour une installation ou une mise à jour, vous pouvez vous rendre sur différents sites de téléchargement. La version de WAMP 5 utilisée lors de l'écriture de cet ouvrage est la 1.4.4.

Les explications et les copies écran qui suivent concernent cette version. Les opérations à mener pour des versions plus récentes peuvent être différentes.

- Double cliquez sur l'icône. L'installation se fait automatiquement. Vous pouvez laisser les options par défaut.



- Une fois l'installation terminée, l'application est lancée et son icône apparaît dans la barre des tâches.



L'icône doit être entièrement blanche. Dans le cas contraire, certains services n'ont pu être chargés correctement. Il faut alors relancer tous les services en effectuant un clic sur l'icône et en choisissant **Restart All Services**. Si le problème persiste, quittez WAMP en effectuant cette fois-ci un clic droit et en choisissant **Exit** puis relancez l'application.

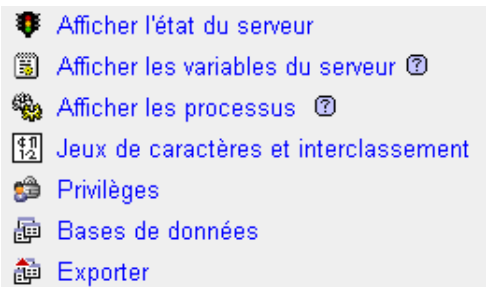
L'application SA Cagou ne comportant pas de pages Web et d'applets, le serveur Apache peut être actif ou arrêté sans provoquer de dysfonctionnements.

Lors de la première utilisation, l'accès à MySQL ne comporte pas de mot de passe. Nous allons en ajouter un.

- Effectuez un clic sur l'icône dans la barre des tâches et choisissez **phpMyAdmin**. WAMP vous adresse un message.

"Votre fichier de configuration fait référence à l'utilisateur root sans mot de passe, ce qui correspond à la valeur par défaut de MySQL. Votre serveur MySQL est donc ouvert aux intrusions, et vous devriez corriger ce problème de sécurité."

- Sur la page principale cliquez sur **Privilèges**.



A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [Tout afficher]

	Utilisateur	Serveur	Mot de passe	Privilèges globaux	"Grant"	
<input type="checkbox"/>	N'importe quel	localhost	Non	ALL PRIVILEGES	Oui	
<input type="checkbox"/>	root	localhost	Oui	ALL PRIVILEGES	Oui	

Veillez noter que les noms de privilèges sont exprimés en anglais

☐ Tout cocher / Tout décocher

Changer les privilèges

- Cliquez sur l'icône **Changer les privilèges** en face de **root**. Saisissez votre mot de passe. Celui utilisé dans l'application SA Cagou est "tempo", en minuscule (sans les guillemets).

• Modifier le mot de passe

☐ aucun mot de passe

☐ Mot de passe:

Entrer à nouveau:

Exécuter

Après la création du mot de passe, un message apparaît.

Le mot de passe de 'root'@'localhost' a été changé.

requête SQL:

```
SET PASSWORD FOR 'root'@'localhost' = PASSWORD( '*****' )
```

[[Modifier](#)] [[Créer source PHP](#)]

Les paramètres de l'utilisateur **root** ont été modifiés. Il vous faut mettre à jour ces données dans le **config.inc.php**.

- Ouvrez le fichier **config.inc.php**. Il est habituellement dans le dossier suivant : **C:\wamp\www\phpmyadmin**.
- Recherchez la ligne suivante : **\$cfg['Servers'][\$i]['password']** et effectuez les modifications.

config.inc.php Avant	config.inc.php Après
<code>\$cfg['Servers'][\$i]['password'] = '';</code>	<code>\$cfg['Servers'][\$i]['password'] = 'tempo';</code>

Ou mettez votre propre mot de passe à la place de "tempo".

- Relancez le serveur MySQL en cliquant sur **Restart Service**.

Création de la base de données MySQL

MySQL gère plusieurs types de tables : **MyISAM**, **InnoDB** et **Heap**. Nous retenons le type MyISAM pour sa stabilité et sa facilité de mise en œuvre.

La base de données de l'application SA Cagou est composée de quatre tables : client, article, commande, ligne_commande. Nous allons les créer rapidement à l'aide de **phpMyAdmin** en limitant le nombre de champs au strict minimum.

- Effectuez un clic sur l'icône de WAMP et choisissez **phpMyAdmin**.
- Assurez-vous que (**Base de données**)... est visible dans la liste déroulante à gauche puis cliquez sur :



- Dans le champ **Créer une base de données**, saisissez **bdjavacagou**.

MySQL vous propose ensuite de créer les tables.

- Créez la table client selon les données ci-après en choisissant l'onglet **Structure** :

Créer une nouvelle table sur la base **bdjavacagou**:

Nom:

Champs:

Champ	Type ?	Taille/Valeurs*	Null	
<input type="text" value="Code"/>	<input type="text" value="VARCHAR"/>	<input type="text" value="5"/>	<input type="text" value="not null"/>	<input checked="" type="radio"/>
<input type="text" value="Nom"/>	<input type="text" value="VARCHAR"/>	<input type="text" value="20"/>	<input type="text" value="not null"/>	<input type="radio"/>
<input type="text" value="Type"/>	<input type="text" value="VARCHAR"/>	<input type="text" value="20"/>	<input type="text" value="null"/>	<input type="radio"/>

...

Type de la table:

Le champ **Code** est déclaré clé primaire, seul le champ **Type** est autorisé à avoir une valeur nulle. Les champs **Code** et **Nom** peuvent recevoir des chaînes de caractères de longueur variables. Le type de table retenu est MyISAM.

- Cliquez sur le bouton **Exécuter**.

Champ	Type
Code	varchar(5)
Nom	varchar(20)
Type	varchar(15)

client

- Créez les autres tables selon les structures suivantes :

Champ	Type
<u>Code</u>	varchar(8)
Désignation	varchar(20)
Descriptif	text

article

Champ	Type
<u>Numero</u>	int(11)
Date	date
Code_Client	varchar(5)

commande

Champ	Type
<u>Num_Commande</u>	int(11)
<u>Code_Article</u>	varchar(8)
Quantite	int(11)

ligne_commande



Veillez à bien déclarer comme clés primaires, les champs **Num_Commande** et **Code_Article** de la table **ligne_commande**. Ce sont des clés étrangères permettant d'établir le lien avec les tables **commande** et **article**. Pour la table **commande**, indexez le champ **Code_client**. Les index permettent d'accélérer les recherches et les jointures.

Évitez les accents, et par la suite l'orthographe mais aussi la casse des tables et des champs devront être rigoureusement respectées dans le code Java (il en est de même pour PHP).



Les contraintes d'intégrité référentielle ne sont pas abordées ici. Pour en bénéficier, il faut choisir pour les tables le type InnoDB et passer par l'éditeur SQL pour leur création. Exemple :

```
CREATE TABLE commande (
  Num_Commande INT NOT NULL,
  Date DATE,
  Code_Client VARCHAR(5) NOT NULL,
  PRIMARY KEY(Num_Commande),
  FOREIGN KEY(Code_Client) REFERENCES client(Code) ON DELETE CASCADE ON
  UPDATE CASCADE) TYPE = InnoDB;
```

Cette requête fait l'hypothèse que la table client est aussi au format InnoDB. Il est possible de convertir des tables ISAM au format InnoDB et d'utiliser dans une même base de données différents types pour les tables, ce qu'il vaut mieux toutefois éviter.

Nous allons établir les relations entre les tables.

- Cliquez sur le nom de la table **commande** à gauche puis sur :

 [Gestion des relations](#)

- Précisez la clé primaire de la table maître pour la clé étrangère.

Relié à	
Relations internes	
Num_Commande	--
Date	--
Code_Client	client->Code
<input type="button" value="Exécuter"/>	

Faites de même pour la table **ligne_commande** pour les champs **Num_Commande** et **Code_Article**.

Requêtes SQL

Nous allons ajouter quelques enregistrements par table puis interroger la base de données à l'aide de requêtes SQL :

	BO1	BAUMIER SA	Société
client	DR1	DROUX	Particulier
	LA1	LANGLE	Artisan
	ACL1	clavier	
article	AEC1	écran	
	APO1	portable	
commande	11	BO1	14/05/08
	22	LA1	15/05/08
	11	ACL1	5
ligne_commande	11	AEC1	5
	22	ACL1	1


La commande 11 concerne le client BAUMIER SA qui a commandé 5 claviers et 5 écrans. Le client LANGLE n'a commandé qu'un clavier et le client DROUX n'a encore rien commandé.

- Pour ajouter des enregistrements, cliquez sur le nom de la table à gauche puis sur l'onglet **Insérer**. Saisissez les valeurs puis cliquez sur le bouton **Exécuter**.

Nous voulons la liste des clients.

- Cliquez sur le nom de la table **client** à gauche, saisissez la requête suivante dans l'éditeur SQL puis cliquez sur **Exécuter**.

```
SELECT * FROM client
```

 MySQL encadre systématiquement les champs par des "quotes" que l'on peut obtenir en appuyant simultanément sur les touches [Alt Gr] et la touche **7-à-`**. Ces caractères sont facultatifs. Par contre, les chaînes de caractères et les dates doivent être encadrées par des guillemets ou des apostrophes (à ne pas confondre avec les "quotes").


Quelques autres exemples de requêtes de sélection :

- Pour obtenir la liste des clients qui sont des particuliers :

```
SELECT * FROM client WHERE Type = "Particulier"
```

- Pour obtenir la liste des clients qui ont passé des commandes :

```
SELECT * FROM client, commande  
WHERE Code = Code_Client
```

 La deuxième requête nécessite une jointure entre les deux tables à partir de la clé primaire de la table **client** et de la clé étrangère de la table **commande**. Les enregistrements

affichés sont ceux dont les valeurs de la clé primaire sont égales à celles de la clé étrangère.

Pour finir nous testons une requête "action" qui modifie l'état de la base. Celle-ci double le nombre de claviers commandés par le client LANGLE.

- Dans l'éditeur SQL, saisissez la requête suivante :

```
UPDATE ligne_commande SET Quantite = Quantite * 2 WHERE Num_Commande = 22
```



Pour de plus amples informations sur les requêtes SQL, consultez la documentation de MySQL.

Installation du pilote ODBC pour MySQL

ODBC, *Open Database Connectivity*, gère de nombreux pilotes permettant d'établir la communication entre des applications clientes et des SGBD. Ce n'est pas la solution la plus performante mais elle présente l'avantage de la simplicité. Elle est par ailleurs disponible gratuitement sur pratiquement toutes les plates-formes.

La procédure d'installation est très simple.

- Téléchargez le driver ODBC pour MySQL :

<http://dev.mysql.com/downloads/connector/odbc/3.51.html>

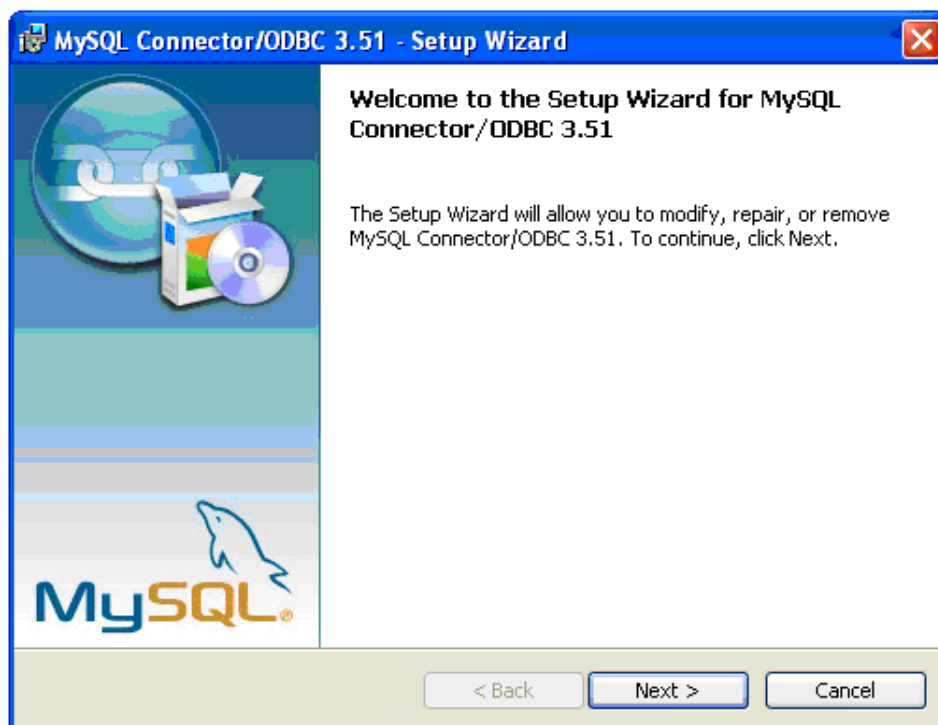
- Décompressez le fichier puis lancez le setup.



mysql-connector-odbc-3.51.1
3 540 Ko



Setup.exe

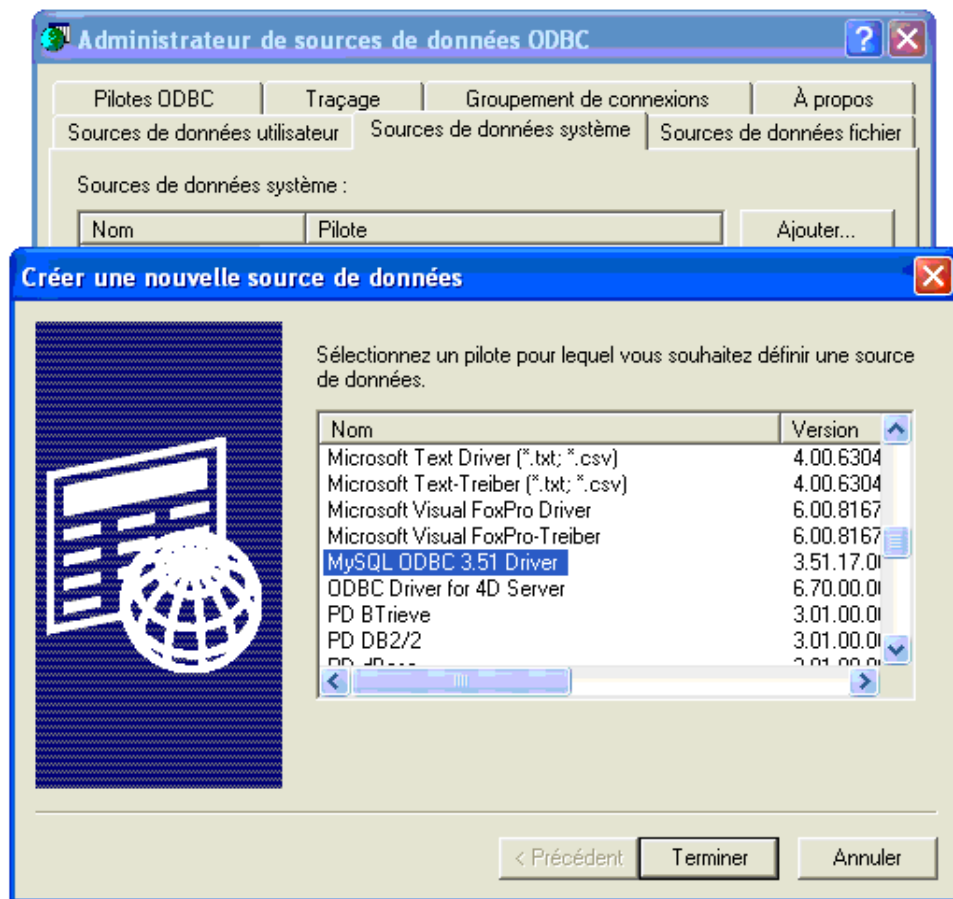


- Lancez le programme ODBC de Windows (**Panneau de configuration - Outils d'administration**).



odbcad32.exe

- Cliquez sur l'onglet **Source de données système** puis sur le bouton **Ajouter...** Choisissez le driver pour MySQL que vous venez d'installer.

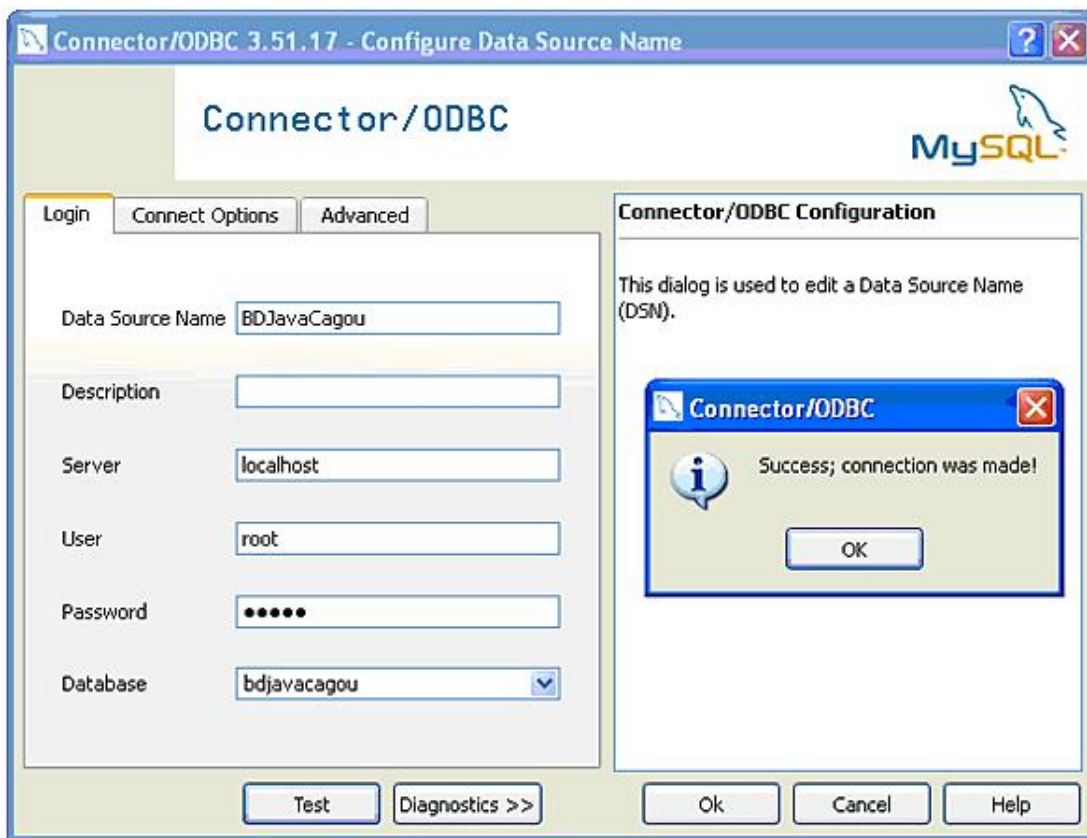


- Configurez les paramètres de la source de données.

Le nom donné au **Data Source Name** doit être explicite et ne pas comporter d'espace. Par commodité pour le développement Java que nous allons réaliser, conservez le nom proposé ci-après.

Pour le serveur, saisissez **localhost**. Le DSN établit la correspondance entre le nom et l'adresse IP du serveur. Vous pouvez le tester en tapant à la place **127.0.0.1**.

Pour le **User** et le **password**, il faut reporter ce qui a été retenu lors de l'installation de WAMP : **root** et **tempo**.

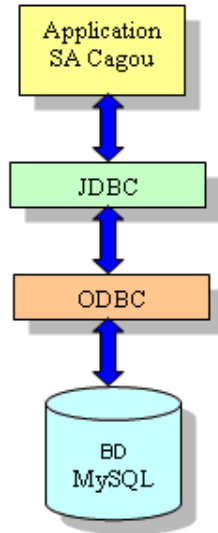


- Assurez-vous que WAMP est actif puis testez la connexion.

JDBC

JDBC, *Java Database Connectivity*, permet à des applications clientes développées en Java d'accéder à des bases de données relationnelles. C'est une API pour laquelle il existe quatre types de pilotes dont les performances et la portabilité sont variables.

Pour notre projet, nous utilisons le type 1 basé sur ODBC. Les appels JDBC sont dans ce cas convertis en appels ODBC. Celui-ci les transmet au SGBDR qui traite les requêtes SQL reçues. De même, les jeux d'enregistrements retournés utilisent la passerelle ou pont JDBC/ODBC entre l'application Java et la base de données.



Les principales opérations réalisées sont les suivantes :

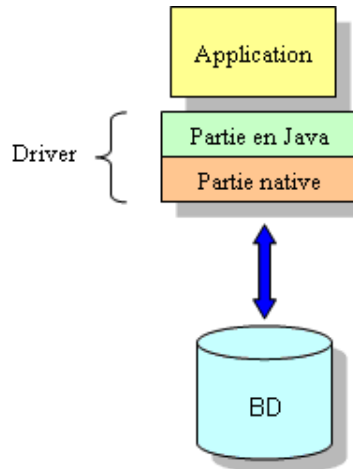
1. Chargement du pilote
2. Établissement de la connexion via le pont JDBC/ODBC avec la base de données
3. Exécution des requêtes SQL par le SGBDR MySQL
4. Récupération des jeux d'enregistrements (pour des requêtes de sélection)
5. Fermeture des jeux d'enregistrements (si existants)
6. Fermeture de la connexion

JDBC fournit toutes les classes utiles pour gérer ces opérations. Nous verrons précisément dans la section Gestion de la connexion du chapitre Développement, comment les utiliser pour exploiter la base de données MySQL.

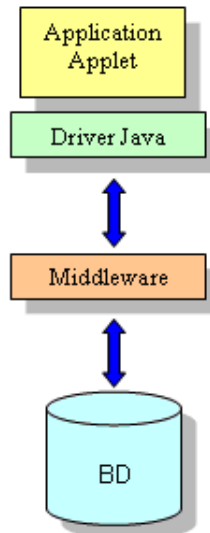
Autres types de pilotes

Il existe trois autres types de pilotes JDBC :

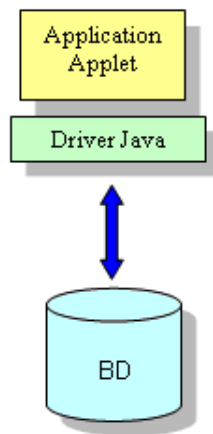
- **Type 2** : pilote dont une partie est écrite en java et l'autre dans le langage du SGBDR. Les appels JDBC sont convertis en appels natifs. Son utilisation nécessite l'installation de la partie native sur le client. Autre inconvénient : la perte de la portabilité du code ce qui exclut son utilisation avec des applets.



- **Type 3** : pilote totalement écrit en Java communiquant avec le SGBD via une application middleware. La portabilité et l'utilisation avec les applets sont donc assurées.



- **Type 4** : pilote totalement écrit en Java avec un accès direct au SGBD. Comme pour le type précédent, la portabilité et l'utilisation avec les applets sont assurées. Ce sont les éditeurs de SGBD qui fournissent ce type de pilotes.



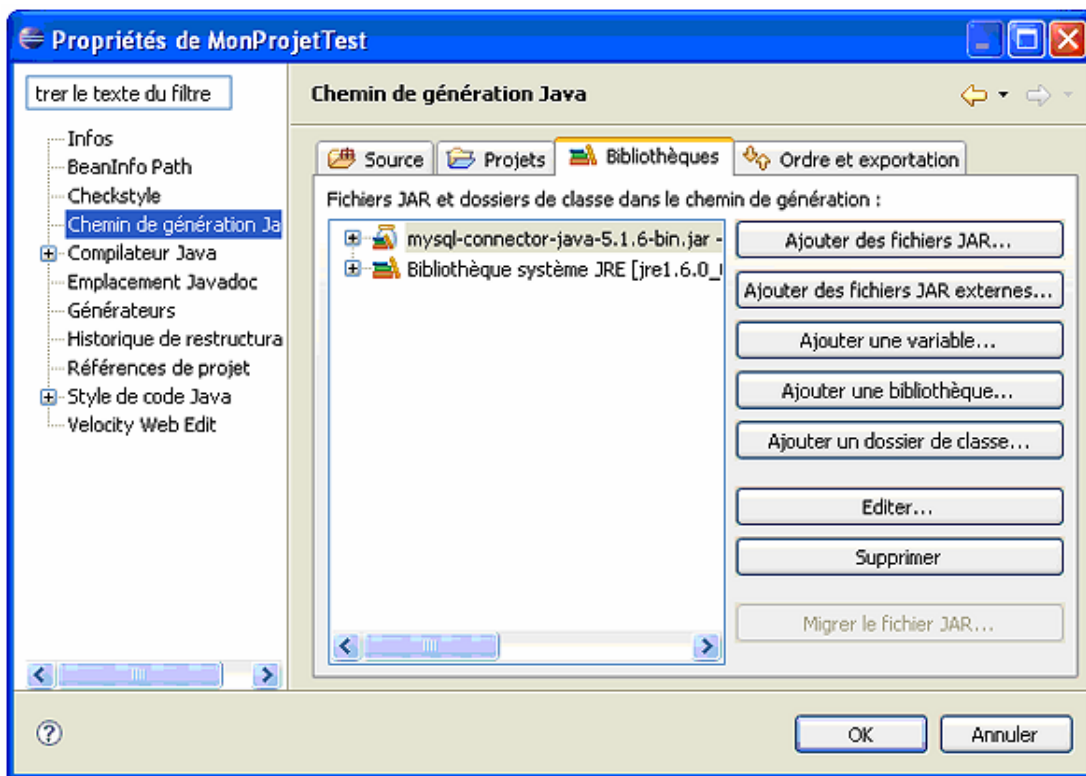
De nombreux drivers sont disponibles sur le site de Sun (<http://java.sun.com/>) dans la section Downloads. Pour accéder rapidement aux drivers, tapez dans le champ de recherche "drivers for MySQL".

Browse All	
JDBC™ API version:	Any ▼
Vendor Name	<input type="text"/>
Certified for J2EE™: <small>(help)</small>	<input type="checkbox"/> J2EE 1.2 <input type="checkbox"/> J2EE 1.3 <input type="checkbox"/> J2EE 1.4 <input type="checkbox"/> Java EE 5 All ▼
Driver type(s): <small>(help)</small>	<input type="checkbox"/> 1 <input type="checkbox"/> 2 <input type="checkbox"/> 3 <input type="checkbox"/> 4 All ▼
Supported DBMS(s):	<div style="border: 1px solid gray; padding: 5px; min-height: 100px;"> IBM AS/400 IDMS IMS Image/Turboimage ImageSQL Informix </div> Match All ▼ of my selections
Required features:	<input type="checkbox"/> Conn.Pooling <input type="checkbox"/> DataSource <input type="checkbox"/> Dist.Trans. <input type="checkbox"/> RowSets
Return how many results per page:	<input type="text" value="20"/>
Search Reset	

Voici l'adresse complète pour accéder à la page **MySQL Connector/ODBC 5.1 Downloads**.

<http://dev.mysql.com/downloads/connector/odbc/5.1.html>

- Pour installer un driver de types 2,3 ou 4, décompressez le fichier jar dans un dossier.
- Sous Eclipse, effectuez un clic droit sur votre projet et choisissez **Propriétés**.
- Sélectionnez ensuite **Chemin de génération Jar** et l'onglet **Bibliothèques**.
- Cliquez sur le bouton **Ajouter des fichiers Jar externes...** et procédez à l'ajout.



- Cliquez ensuite sur l'onglet **Ordre et exportation**, cochez la case du driver puis validez.

☒ mysql-connector-java-5.1.6-bin.jar - C:\Java Eclipse\Connector J 5.1\mysql-connector-java-5.1.6

Classe

La génération spontanée n'existe pas. La voiture que vous conduisez, la chaise sur laquelle vous êtes assis(e) ont été fabriquées. Tous ces objets proviennent d'une "fabrique" ou usine. Pour une première introduction à la notion de classe, ce dernier terme peut être retenu. Il existe ainsi des usines pour avions, bateaux, jouets, etc.

Une autre particularité des classes est le principe de regroupement. On évite dans une même usine de fabriquer à la fois des voitures et des bateaux, ou des jouets et des ordinateurs. Les objets fabriqués appartiennent donc à des catégories qui sont les classes en POO. Nous pouvons retenir qu'une classe est une sorte d'usine qui fabrique des objets qui ont des caractéristiques communes (les bateaux ont une coque, les voitures une carrosserie).

Nous avons vu lors de l'analyse, leur représentation graphique UML. Voyons maintenant précisément leur écriture en Java au travers d'exemples très simples.

Pour déclarer une classe Chat, il suffit d'utiliser le mot clé **Class**.

```
class Chat {  
}
```

Propriétés

La classe Chat est ensuite complétée selon les besoins de chacun. Supposons que nous désirons juste caractériser les chats par leur nom, couleur et âge. Il faut alors ajouter ces caractéristiques ou propriétés à la classe.

```
class Chat {  
    String caractere;  
    String nom;  
    String couleur;  
    int age;  
}
```

Toute propriété ou attribut doit impérativement être typée : entier pour l'âge et chaîne de caractères pour les autres dans notre exemple. Ces types sont simples. On parle de types élémentaires ou primitifs. Vous aurez l'occasion lors du développement d'utiliser des types plus riches notamment des types classes. Les propriétés définissent l'aspect structurel d'une classe.

Méthodes

Les méthodes représentent les actions que peuvent mener des objets issus d'une classe. Elles définissent l'aspect dynamique ou comportemental de la classe. Les chats peuvent dormir, manger, miauler... Ajoutons quelques-unes de ces méthodes.

```
class Chat {
    // propriétés
    String caractere;
    String nom;
    String couleur;
    int age;
    // méthodes
    void dormir(){
    }
    void manger(){
    }
    String miauler(){
        String leMiaulement = "miaou";
        return leMiaulement;
    }
}
```

Les méthodes qui ne renvoient aucune valeur correspondent à des procédures. Elles sont toujours précédées du mot clé **void**.

Les méthodes qui renvoient une valeur correspondent à des fonctions. Elles sont toujours précédées du type de la valeur renvoyée.

Les méthodes peuvent bien sûr contenir des paramètres qui doivent tous être typés.

Parmi les méthodes, certaines ont pour rôle de retourner les valeurs des propriétés et d'autres de les modifier. Elles sont nommées respectivement **accesseurs** et **mutateurs**.

```
class Chat {

    // propriétés
    ...

    // méthodes
    String getNomChat() {
        return nom;
    }
    void setNom(String vNom) {
        nom = vNom;
    }
    String getCouleur() {
        return couleur;
    }
    void setCouleur(String vCouleur) {
        couleur = vCouleur;
    }
    int getAge() {
        return age;
    }
    void setAge(int vAge) {
        age = vAge;
    }
    ...
}
```



Le terme de services est parfois utilisé pour désigner les méthodes et celui d'opération pour l'implémentation des méthodes.

Accessibilité

La POO ajoute le principe d'accessibilité (visibilité) qui consiste à contrôler l'accès aux classes et à leurs membres (propriétés et méthodes). Les niveaux de visibilité sous Java sont les suivants :

1. Membres

- **public** : les membres sont utilisés sans aucune restriction par toute classe et en tout lieu. Pour protéger les propriétés de modifications non autorisées, il est cependant fortement recommandé de ne pas les déclarer publiques.
- **private** : c'est le niveau le plus élevé de restriction. Les propriétés et méthodes ne sont alors accessibles que depuis la classe elle-même. Concernant les propriétés, elles peuvent l'être en dehors de la classe mais uniquement par le biais de ses mutateurs et accesseurs déclarés publics.
- **protected** : les membres sont accessibles aux classes descendantes (voir la notion d'héritage plus loin) mais aussi aux classes non descendantes du même package.

2. Classes

- **public** : en java, les classes sont publiques par défaut. Il convient cependant de le préciser par le mot clé public. Son omission rend par ailleurs les classes inaccessibles en dehors de leur package.
- **protected** : ce niveau ne concerne que les classes internes. Ce sont des classes définies à l'intérieur d'une classe n'étant utilisées que par celle-ci.

D'autres mots clés permettent de caractériser les classes et leurs membres dans des domaines autres que celui de l'accessibilité tels que la sécurité ou l'optimisation.

```
public class Chat {  
    // propriétés  
    private String nom;  
    private String couleur;  
    private int age;  
  
    // méthodes  
    public String getNomChat() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
    public String getCouleur() {  
        return couleur;  
    }  
    public void setCouleur(String couleur) {  
        this.couleur = couleur;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public void dormir(){  
    }  
  
    public void manger(){  
    }  
}
```

```
public String miauler(){  
    String leMiaulement = "miaou";  
    return leMiaulement;  
}  
}
```


Encapsulation

Le fait de regrouper au sein d'une même classe propriétés et méthodes, et de définir le niveau d'accessibilité constitue l'**encapsulation**, un des concepts essentiels de la POO. Seules sont visibles les méthodes publiques, ce qui renforce la sécurité des données et la maintenance des programmes. De plus, l'implantation de la méthode est masquée. Finalement avec le concept d'encapsulation, la classe ne présente qu'une interface composée des méthodes publiques et de leurs éventuels paramètres.

Si nous reprenons notre exemple, pour modifier l'âge d'un chat, il faut utiliser la méthode **setAge()**. En partant de ce principe, on peut l'appliquer à d'autres cas tel que le débit d'un compte qui ne peut se faire sans la méthode appropriée et habilitée. Les risques d'erreur sont ainsi réduits et la sécurité accrue. La modification du code traitant du miaulement d'un chat ou du débit d'un compte n'a que peu voire aucune incidence sur un programme puisque l'implémentation des méthodes est masquée. Exemple :

```
// méthode modifiée
public String miauler(){
    // un chat qui n'arrête pas de miauler
    String leMiaulement = "miaou miaouuuu miaouuuu ...";
    return leMiaulement;
}

// utilisation dans un programme
leChat.miauler();
```

Constructeur

Pour qu'une classe soit complète, il lui faut un constructeur. Il s'agit d'une méthode particulière qui permet de créer des objets. Si vous omettez de définir un constructeur, Java le fera pour vous au moment de la compilation du programme en créant un constructeur basique du type **nomDeLaClasse()**. Ajoutons un constructeur à notre classe. Exemple :

```
public class Chat {  
    // propriétés  
    ...  
    // constructeur  
    public Chat(String leNom, String laCouleur, int sonAge){  
        nom = leNom;  
        couleur = laCouleur;  
        age = sonAge;  
    }  
    // méthodes  
    ...  
}
```

Le constructeur porte le même nom que la classe. Il est toujours public sinon comment les objets pourraient-ils être créés ? Le constructeur **Chat()** attend trois paramètres ce qui permet de créer un chat avec un nom, une couleur et un âge. Exemple dans un programme :

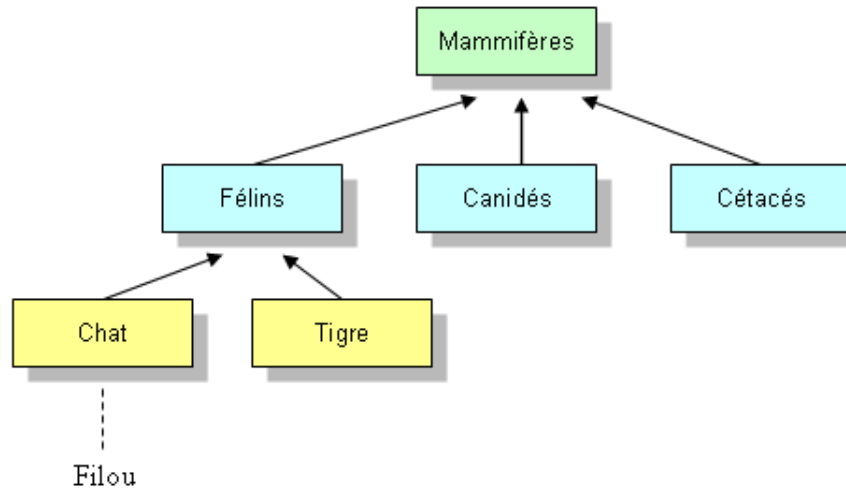
```
...  
Chat leChat = new Chat("Filou", "rouquin", 2);  
...
```



Certaines classes particulières n'ont pas besoin de constructeurs, ce sont les interfaces. Elles sont constituées uniquement de méthodes abstraites.

Héritage

L'héritage est un concept aisé à comprendre. Par exemple, nous héritons de nos parents certains traits physiques et peut-être certains comportements. Tel père tel fils, dit-on souvent. En POO, l'idée est la même. Si on transpose le concept d'héritage aux classes, cela revient à définir une classe générique dont dérivent d'autres classes. Pour la classe Chat, il est possible de la rattacher à la classe mammifère. Selon la partie du monde réel des animaux que l'on désire représenter, une structure de classes est créée. Exemple :



Tous les félins, canidés et cétacés héritent de la classe **Mammifères** des caractéristiques communes : animaux à sang chaud, la femelle allaite ses petits, etc. Filou fait partie d'une grande famille. Il va hériter des classes ancêtres toutes les propriétés et méthodes de celles-ci sauf celles déclarées privées. Pour en tenir compte, il faut revoir la définition de la classe **Chat** et ajouter celles des classes parentes. Pour faire simple, voici un exemple d'héritage limité aux classes **Félin** et **Chat**.

```
public class Felin {
    private String espece;
    private String couleur;
    public Felin(String espece, String couleur) {
        this.espece = espece;
        this.couleur = couleur;
    }
    // les mutateurs et accesseurs
    public String getCouleur() {
        return couleur;
    }
    public void setCouleur(String couleur) {
        this.couleur = couleur;
    }
    public String getEspece() {
        return espece;
    }
    public void setEspece(String espece) {
        this.espece = espece;
    }
}

public class Chat extends Felin {
    private String nom;
    private int age;
    public Chat(String vEspece, String vCouleur, String vNom, int vAge) {
        super(vEspece, vCouleur);
        nom = vNom;
        age = vAge;
    }
    // les mutateurs et accesseurs
    ...
    // les autres méthodes
}
```

```
...  
}
```

Le mot clé **extends** indique que la classe dérivée (fille) **Chat** étend la classe de base (mère) **Felin**. L'appel du constructeur de la classe mère s'effectue avec le mot clé **super**.

Redéfinition des méthodes

Les méthodes de la classe mère peuvent être redéfinies dans les classe filles ou sous-classes. Il s'agit en fait d'une spécialisation. Dans le sens inverse, c'est une généralisation qui permet de factoriser les comportements communs. Par exemple, tous les mammifères se nourrissent mais les chats domestiques mangent des croquettes ce qui n'est pas forcément le cas des tigres. Voyons ce que cela donne avec nos classes :

```
public class Felin {  
    ...  
    public void manger(String vNourriture){  
    }  
}  
  
public class Chat {  
    ...  
    public void manger(String vNourriture){  
        System.out.println("Je mange " + vNourriture);  
    }  
}
```

Exemple dans un programme :

```
...  
leChat.manger("croquettes")
```

Polymorphisme

Dans une première approche, le polymorphisme correspond à la capacité d'un objet à choisir selon son type (sa classe) au moment de l'exécution (dynamiquement) la méthode qui correspond le mieux à ses besoins parmi ses propres méthodes ou celles des méthodes mères. Une autre particularité du polymorphisme est que toutes les méthodes concernées portent le même nom. Au sein d'une classe, elles se différencient obligatoirement par le nombre de paramètres ou le type de ces derniers.

En ajoutant à la structure de classes précédente, la classe Baleine, nous avons à définir ce que mange les baleines avec une méthode se nommant également **manger()**. Modifions aussi la classe Chat pour mettre en évidence le polymorphisme de méthodes. Exemple :

```
public class Baleine {
    ...
    public void manger(String vNourriture){
        System.out.println("Je mange du " + vNourriture + ". Qui suis-je ?");
    }
}

public class Chat {
    ...
    public String ronronner(){
        String leMiaulement = "ron ron ...";
        return leMiaulement;
    }
    public void manger(String vNourriture){
        System.out.println("Je mange des " + vNourriture);
        System.out.println("Quand j'ai bien mangé, je ronronne " + ronronner());
    }
}
```

Exemple dans un programme :

```
...
leChat.manger("croquettes");
laBaleine.manger("plancton");
```

À l'exécution :

```
Je mange du plancton. Qui suis-je ?
Je mange des croquettes
Quand j'ai bien mangé, je ronronne ron ron ...
```

La deuxième forme de polymorphisme correspond à la capacité d'un objet à changer de type, autrement dit de classe. Cela consiste à sous-typer une propriété de la classe mère.

Démarche

Les tâches préparatoires ont été réalisées : personnalisation d'Eclipse avec l'ajout de plugins, analyse du projet avec l'élaboration des principaux diagrammes UML, paramétrage du serveur Wamp et création de la base de données MySQL. Le développement peut commencer.

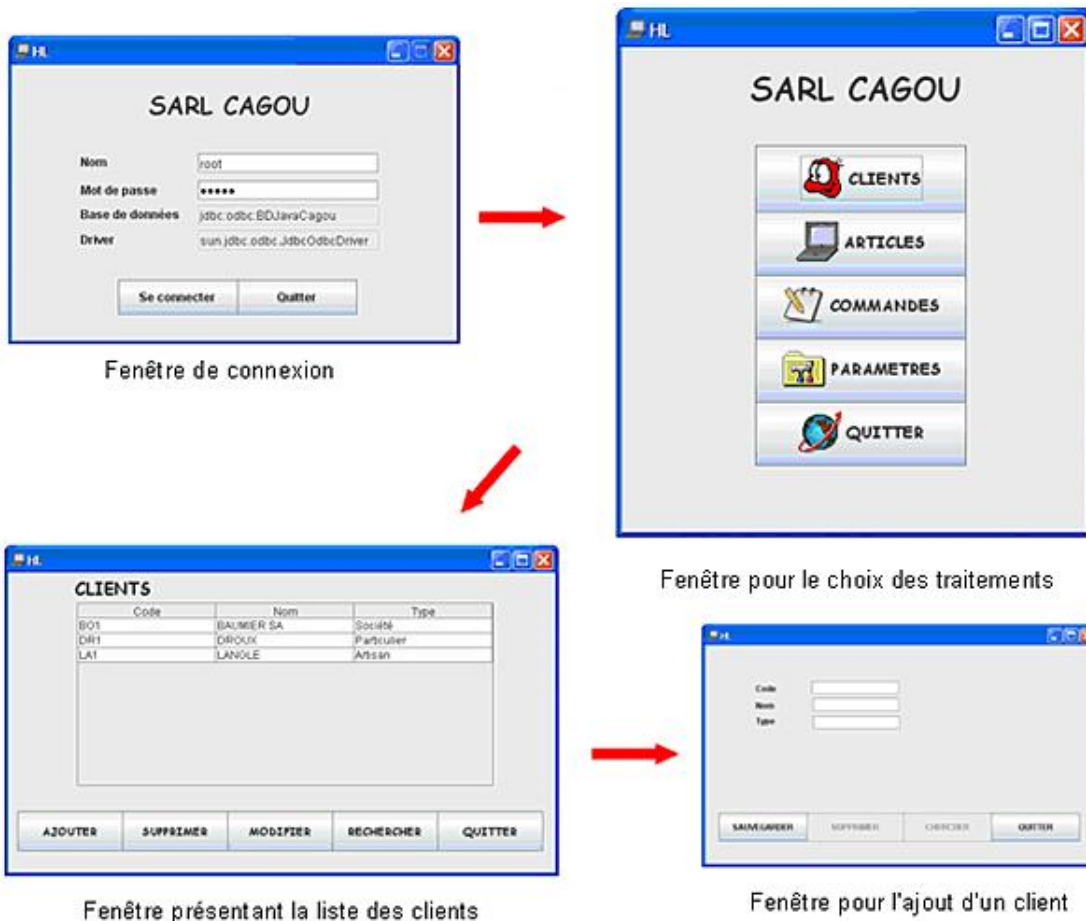
Tel qu'annoncé dans le chapitre Analyse, nous produisons en premier lieu les maquettes de la future application. Pour la gestion des clients, nous avons recensé cinq fenêtres :

- connexion : saisie du nom et du mot de passe.
- menu : choix du fichier à traiter.
- table : présentation des enregistrements au format table avec choix des traitements :

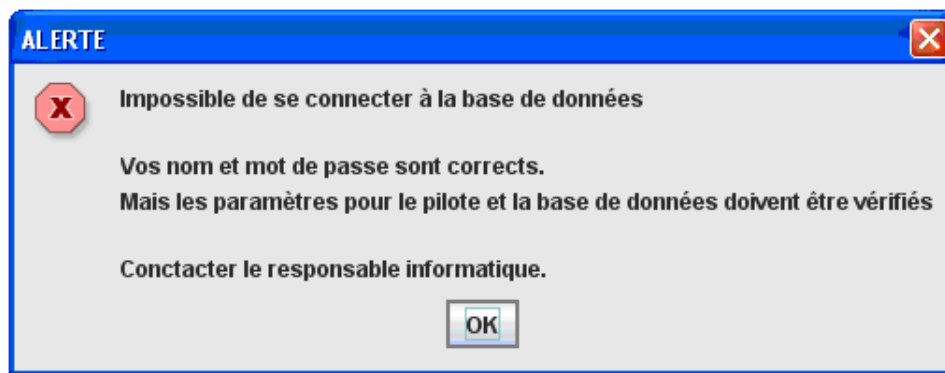
AJOUTER	SUPPRIMER	MODIFIER	RECHERCHER	QUITTER
---------	-----------	----------	------------	---------


- fiche : présentation des données pour un enregistrement au format fiche selon le traitement choisi.
- recherche : fenêtre dédiée à la recherche et présentant les enregistrements en mode table.

Pour coder chaque fenêtre, nous partons des classes graphiques du diagramme des classes. Nous assurons ensuite l'enchaînement des fenêtres sans nous soucier à ce stade des traitements. Reprenons l'exemple pour l'ajout d'un client déjà vu au chapitre Présentation du projet.



Nous complétons ensuite les autres classes et effectuons enfin la jonction entre elles en nous référant à l'analyse réalisée. Nous ajoutons à ce moment-là, toute boîte de dialogue utile à l'information de l'utilisateur. Exemple :



 La démarche est identique pour les articles et les commandes.

Fenêtre de connexion

Nous avons vu au chapitre Concepts de base de la POO, les concepts de base de la programmation orientée objet. Il s'agit maintenant de les mettre en œuvre au travers de notre projet. Si certaines notions vous semblent encore floues, reportez-vous au chapitre en question.

1. Fonctionnalités

Considérons les différentes étapes :

1. Présentation de la fenêtre de connexion.
2. Saisie des paramètres de connexion.
3. Validation ou abandon.

L'utilisateur est accueilli par une fenêtre dans laquelle il doit saisir son nom et son mot de passe.

Maquette de la fenêtre de connexion 'HL'. La fenêtre a un titre 'HL' et des boutons de gestion de fenêtre (minimiser, maximiser, fermer). Elle contient quatre champs de saisie étiquetés 'Nom', 'Mot de passe', 'Base de données' et 'Driver'. Les valeurs saisies sont 'root', '*****', 'jdbc:odbc:BDJavaCagou' et 'sun.jdbc.odbc.JdbcOdbcDriver' respectivement. En bas, il y a deux boutons : 'Se connecter' et 'Quitter'.

Dans la maquette, les contrôles de saisie sont absents et les boutons ne sont pas opérationnels. Nous nous contentons d'afficher de simples messages suite à une action de l'utilisateur comme un clic de souris (la gestion des événements sera abordée plus loin dans ce chapitre).

Exemples :

Maquette de message de connexion établie. La fenêtre a un titre 'Maquette' et un bouton de fermeture. Elle contient un message d'information : 'Connexion établie. Affichage de la fenêtre principale.' et un bouton 'OK'.

Maquette de message d'abandon de connexion. La fenêtre a un titre 'Maquette' et un bouton de fermeture. Elle contient un message d'information : 'Abandon de la connexion.' et un bouton 'OK'.

2. Création de la maquette

Les fenêtres sont issues de classes graphiques. Nous utilisons la classe **FenConnexion** issue du diagramme de classes. Cette classe aborde en particulier les points suivants :

- classes graphiques ;
- héritage.

Toute la partie graphique peut être codée directement. Étant au troisième millénaire et afin de ne pas réinventer la roue sans parler du temps perdu, vous allez utiliser **Visual Editor** installé précédemment (cf. chapitre Environnement de développement).

- Ouvrez la classe **FenConnexion** à partir du paquetage **dialogue** en effectuant un clic droit sur la classe et en choisissant **Ouvrir avec - Visual Editor**.

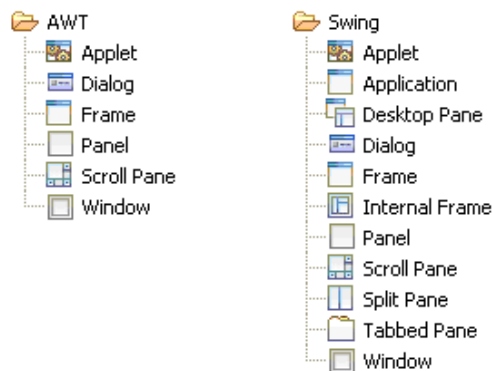
Voyons le code généré par Eclipse-Visual Editor. Les mots clés du langage Java apparaissent en gras dans l'éditeur de texte.

Le paquetage contenant la classe est précisé.

```
package dialogue
```

Java propose plusieurs types de fenêtres issues bibliothèques dédiées aux interfaces graphiques, **AWT** et **Swing**. Certaines classes de la bibliothèque Swing enrichissent celles de même nom de la bibliothèque AWT. Swing est la plus récente et comporte de nombreuses fonctionnalités non disponibles dans AWT. Toutes les classes de Swing commencent par la lettre J.

Ci-après, une représentation des classes graphiques dans Visual Editor.



Lors de l'élaboration du diagramme de classes, nous avons choisi une fenêtre de type **JFrame**. Les classes nécessaires à la construction d'une fenêtre standard de ce type ont été importées.

```
import java.awt.BorderLayout;
import javax.swing.JPanel;
import javax.swing.JFrame;
```

Nous avons vu au chapitre Concepts de base de la POO qu'une classe est déclarée principalement **public** ou **protected** et que ces instructions déterminent l'accessibilité à la classe. La classe **FenConnexion** déclarée **public** est donc accessible partout dans le projet. Pour le vérifier, vous pouvez créer une classe dans un des paquetages ou dans un nouveau paquetage et ajouter le code suivant :

```
public static void main(String[] args) {
    // TODO Raccord de méthode auto-généré
    FenConnexion laFenetre = new FenConnexion();
    laFenetre.setVisible(true);
}
```

L'instruction **extends** précise que la fenêtre hérite de la classe **JFrame** qui comporte des caractéristiques communes à toutes les fenêtres de ce type.

```
public class FenConnexion extends JFrame
```

Positionnement de la classe JFrame :

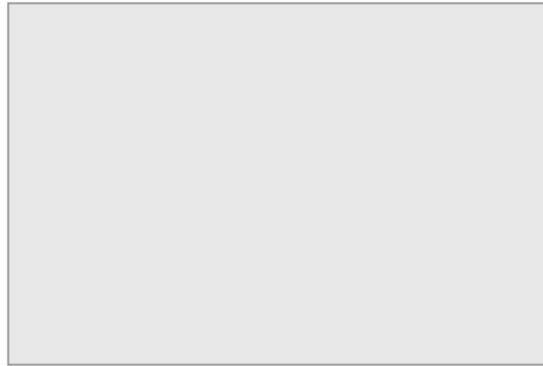
```

java.lang.Object
├── java.awt.Component
│   ├── java.awt.Container
│   │   ├── java.awt.Window
│   │   │   ├── java.awt.Frame
│   │   │   └── javax.swing.JFrame

```

Il existe en Java deux sortes de composants, les lourds et les légers. Les premiers dépendent du système hôte et ne peuvent donc fonctionner que sur ce système. Certains, en très petit nombre sont constitués de code Java s'interfaçant avec les fonctions du système d'accueil. La plupart sont des composants identiques quel que soit le système. Ils communiquent avec le système hôte par le biais d'un élément dépendant de celui-ci réduisant ainsi la communication entre les deux environnements au strict minimum. Les composants légers sont entièrement écrits en Java et peuvent donc être utilisés sur l'ensemble des systèmes.

JFrame est un composant lourd qui étend la classe **Frame** de l'ancienne bibliothèque AWT, elle-même dérivée de la classe **Window**. Les instances de celle-ci sont de simples rectangles sans titre et sans bouton.



Difficile de faire plus dépouillé pour une fenêtre.

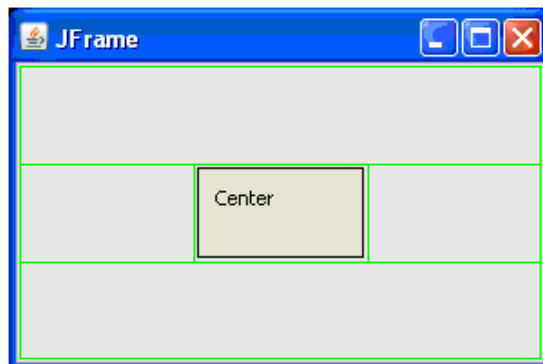
L'arborescence nous montre qu'une **JFrame** est une fenêtre rectangulaire (plus complète) qui est un composant de type conteneur pouvant contenir d'autres composants. On aboutit finalement à la classe **Object**, classe mère de toutes les classes.

Lors de la création de la classe graphique, Eclipse génère automatiquement un identificateur de version par défaut.

```
private static final long serialVersionUID = 1L;
```

La fenêtre est composée d'un panel ou panneau. Si des composants sont ajoutés à la fenêtre, ils sont positionnés dans le panneau. Un panneau peut contenir d'autres panneaux.

Structure d'une fenêtre **JFrame** standard créée avec Eclipse - Visual Editor.



Le composant nommé **JContentPane** par Eclipse est issu de la classe **JPanel**. C'est donc un composant de type panneau pouvant contenir d'autres composants tels que les champs de saisie ou les boutons.

Positionnement de la classe **JPanel**.

```

java.lang.Object
├── java.awt.Component
│   ├── java.awt.Container
│   │   ├── javax.swing.JComponent
│   │   └── javax.swing.JPanel

```

```
private JPanel jContentPane = null;
```

Le constructeur qui porte obligatoirement le même nom que la classe comporte ici deux méthodes : **super()** qui permet d'invoquer le constructeur de la classe mère et **initialize()** qui précise les caractéristiques par défaut de la fenêtre instanciée.

```
public FenConnexion() {
    super();
    initialize();
}
```

La méthode **initialize()** qui initialise la fenêtre lors de l'appel du constructeur.

```
private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
}
```

Profitez-en pour changer le titre en modifiant l'argument de la méthode **setTitle()**.

La méthode **getJContentPane()** appelée par la méthode **initialize()** initialise le panel. Celui-ci ne contient pour l'instant aucun composant ou **widget**.

```
private JPanel getJContentPane() {
    if (jContentPane == null) {
        jContentPane = new JPanel();
        jContentPane.setLayout(new BorderLayout());
    }
    return jContentPane;
}
```

Dans la méthode **main()**, la ligne suivante est inutile car la classe **FenConnexion** étend la classe **JFrame** qui possède déjà la méthode :

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            FenConnexion thisClass = new FenConnexion();
            thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            thisClass.setVisible(true);
        }
    });
}
```

- Supprimez cette ligne :

```
thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

D'autre part, il vaut mieux regrouper les méthodes agissant sur la fenêtre.

- Déplacez la ligne **thisClass.setVisible(true)** de la méthode **main()**, modifiez **thisClass** par **this**. Le mot clé **this** n'est cependant pas obligatoire.

```
this.setSize(300, 200);this.setContentPane(getJContentPane());this.setTitle("JFrame");this.setVisible(true);
```

- Conserver un **handle** sur cette fenêtre ne présente pas d'intérêt réel ici. Nous la refermerons une fois la connexion établie. Aussi, vous pouvez modifier la ligne correspondante.

Code de classe Fenetre_Connexion remanié :

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new Fenetre_Connexion();
        }
    });
}
```

Le plus souvent, une référence ou handle est créé en même temps que l'objet. Cette référence permet ensuite d'utiliser l'objet en mémoire. Perdre le handle, c'est alors perdre la possibilité d'accéder à l'objet. Il est possible de créer un objet sans une référence explicite, par exemple, lors de l'ajout d'un objet à un vecteur ou par le biais d'un autre objet. Il est également possible d'affecter plusieurs handles au même objet et de créer un objet sans qu'un handle ne lui soit affecté.

3. Personnalisation de la maquette

Nous allons maintenant personnaliser cette fenêtre standard avec Visual Editor pour obtenir la fenêtre de connexion.

- Modifiez la propriété **layout** du panneau nommé **jContentPane** afin de pouvoir disposer librement les composants. Sélectionnez celui-ci et effectuez le choix suivant :

Erreurs	Javadoc	Déclaration	Console	Propriétés	Débogage	Font and Colors
Propriété		Valeur				
background		238,238,238				
border						
componentOrientation		UNKNOWN				
enabled		true				
>field name		jContentPane				
font		Dialog, plain, 12				
foreground		51,51,51				
>layout		null				
name		null				
preferredSize		BorderLayout				
toolTipText		BoxLayout(X_AXIS)				
visible		BoxLayout(Y_AXIS)				





La 4^{ème} ligne de la méthode **getJContentPane()** a été modifiée.

```
private JPanel getJContentPane() {
    if (jContentPane == null) {
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
    }
    return jContentPane;
}
```

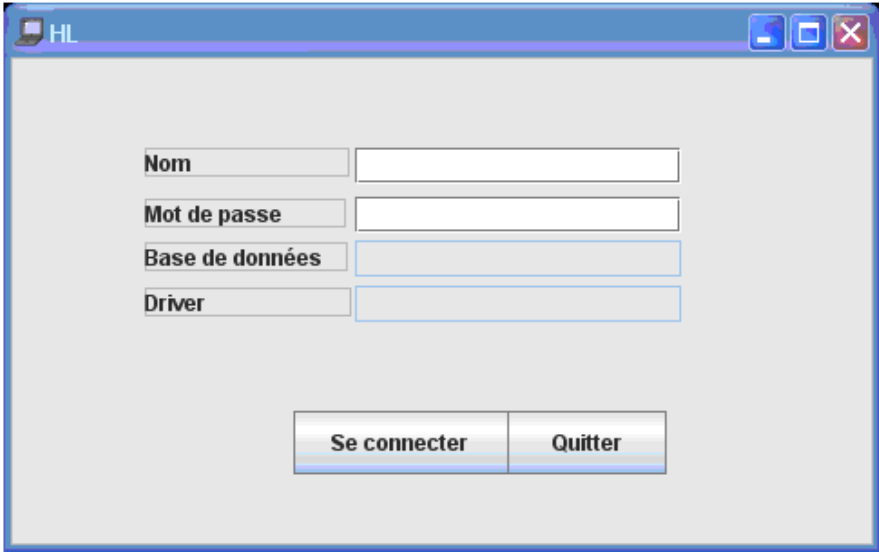
- Cliquez sur **Swing Components** dans la palette sur le côté droit.



- Choisissez les composants suivants :

 JLabel	Etiquette
 JTextField	Champ de saisie
 JPasswordField	Champ de saisie pour les mots de passe
 JButton	Bouton

- Déposez ces composants pour obtenir la fenêtre de connexion en les nommant comme indiqué ci-après.



Labels	Champs de saisie simple ou password
nomUtilisateur_Label	nomUtilisateur_TextField
motDePasse_Label	motDePasse_PasswordField
serverBD_Label	serverBD_TextField
driverSGBD_Label	driverSGBD_TextField

Pour les boutons, nommez-les : **btn_Connexion** et **btn_Quitter**.

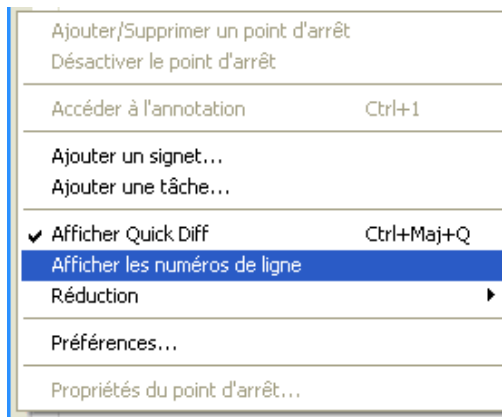
Voyons le code modifié de la classe **Fenetre_Connexion** généré par Eclipse.

```

import java.awt.BorderLayout;
import java.awt.Dimension;
```

Suite aux modifications réalisées, nous constatons que certaines classes importées du paquetage java.awt ne sont plus utiles. Vous pouvez les supprimer.

Nous pouvons constater que l’ajout des composants s’est traduit par de nombreuses lignes de code. Pour afficher le nombre de lignes, effectuez un clic droit sur le bord gauche de l’éditeur de code puis choisissez **Afficher les numéros de ligne**.



La méthode **main()** permet d'exécuter la classe. La portion de code qui suit **SwingUtilities.invokeLater(new Runnable() { ... })** précise ce qui doit être exécuté, en l'occurrence notre fenêtre de connexion après sa création.

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FenConnexion();
        }
    });
}
```

En fait, deux threads sont créés, l'un pour tous les traitements non graphiques, l'autre pour les traitements dépendants de Swing. Nous respectons ainsi la règle qui consiste à séparer les traitements graphiques et non graphiques dans des threads différents.

Il est possible de créer une classe non graphique uniquement pour le lancement de l'application. Exemple :

```
package packageCagou;
import javax.swing.SwingUtilities;

public class Application_SA_Cagou {

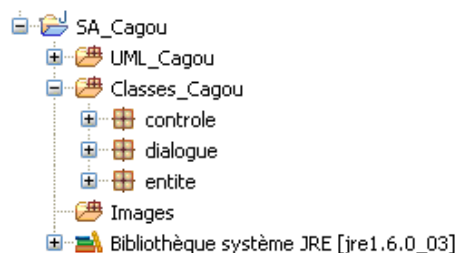
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable(){
            public void run(){
                new FenConnexion();
            }
        });
    }
}
```

Si vous optez pour cette option, pensez à supprimer la méthode **main()** de la classe **FenConnexion**.

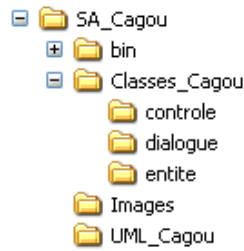
Pour apporter une touche finale à notre première fenêtre, nous désirons personnaliser l'icône et le titre de la fenêtre. Exemple :



- Sous Eclipse, ajoutez un sous-dossier nommé **images** dans le dossier du projet.



- Sous Windows, copiez vos images dans le sous-dossier **images**.



- Dans le code, effectuez l'import de la classe **ImageIcon** du paquetage **javax.swing** et les modifications suivantes. Nous voulons aussi centrer la fenêtre et interdire son redimensionnement.

```
// importation des bibliothèques graphiques
...
import javax.swing.ImageIcon;

/* METHODES
***** */

private void initialize() {this.setSize(448, 315);
    ...
    // pour centrer la fenêtre
    this.setLocationRelativeTo(null);
    // pour interdire le redimensionnement de la fenêtre
    this.setResizable(false);
    // personnalisation
    this.setTitle("SA CAGOU");
    ImageIcon image = new ImageIcon("images\\article.gif");
    this.setIconImage(image.getImage());
    ...
    this.driverSGBD_TextField.setFocusable(false);
}
```

- Testez la classe. Effectuez un clic droit dans le code puis choisissez **Exécutez en tant que - 1 Application Java**. Bien entendu, à ce stade, les boutons ne produisent aucune action.

À l'issue de ces premiers travaux, nous avons créé une interface graphique sommaire qui a toutefois nécessité plus de 150 lignes de code. Fort heureusement, Visual Editor a généré l'essentiel du code facilitant ainsi la création puis la personnalisation de la maquette.

Code complet de la classe **FenConnection** version 1

```
/*
 * FenConnection Version 1 (maquette)
 * boutons inactifs
 * aucun traitement
 */

package dialogue;

//bibliothèques graphiques
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;

import java.awt.Rectangle;
import javax.swing.JButton;
import java.awt.Font;
import javax.swing.SwingConstants;

public class FenConnexion extends JFrame {
    // PROPRIETES
```



```

// *****
private static final long serialVersionUID = 1L;
// le panneau
private JPanel jContentPane ;

private JTextField nomUtilisateur_TextField ;
private JPasswordField motDePasse_PasswordField ;
private JTextField serverBD_TextField ;
private JTextField driverSGBD_TextField ;

private JLabel nomUtilisateur_Label ;
private JLabel motDePasse_Label ;
private JLabel serverBD_Label ;
private JLabel driverSGBD_Label ;

private JButton btn_Connexion ;
private JButton btn_Quitter ;

private JLabel titre_Label = null;

// CONSTRUCTEUR
public FenConnexion() {
    // appel du constructeur de JFrame
    super();
    // initialisation des valeurs propres
    // à la fenêtre connexion (voir ci-dessous)
    initialize();
}

// METHODES
// initialisation de la fenêtre
// - son aspect général
// - avec ses widgets contenus dans le panneau
private void initialize() {
    this.setSize(448, 315);
    // pour centrer la fenêtre
    this.setLocationRelativeTo(null);
    // pour interdire le redimensionnement de la fenêtre
    this.setResizable(false);
    this.setContentPane(getJContentPane());
    // ne pas oublier de rendre la fenêtre visible
    this.setVisible(true);
    // pour faire simple
    // interdire la modification des champs concernant la BD
    this.serverBD_TextField.setEditable(false);
    this.driverSGBD_TextField.setEditable(false);
    // et les rendre non focusables pour faciliter la saisie au clavier
    this.serverBD_TextField.setFocusable(false);
    this.driverSGBD_TextField.setFocusable(false);
}

// la Frame créée contient un composant JPanel <-> panneau, nommé ici jContentPane
// création et initialisation du jContentPane avec ses composants
private JPanel getJContentPane() {
    if (jContentPane == null) {
        titre_Label = new JLabel();
        titre_Label.setBounds(new Rectangle(64, 19, 305, 40));
        titre_Label.setFont(new Font("Comic Sans MS", Font.BOLD, 24));
        titre_Label.setHorizontalAlignment(SwingConstants.CENTER);
        titre_Label.setHorizontalTextPosition(SwingConstants.CENTER);
        titre_Label.setText("SARL CAGOU");
        jContentPane = new JPanel();
        jContentPane.setLayout(null);

        // création et initialisation des labels
        nomUtilisateur_Label = new JLabel();
        nomUtilisateur_Label.setBounds(new Rectangle(69, 88, 113, 16));
        nomUtilisateur_Label.setText("Nom");
        motDePasse_Label = new JLabel();
    }
}

```

```

motDePasse_Label.setBounds(new Rectangle(69, 116, 111, 16));
motDePasse_Label.setText("Mot de passe");
serverBD_Label = new JLabel();
serverBD_Label.setBounds(new Rectangle(69, 140, 112, 16));
serverBD_Label.setText("Base de données");
driverSGBD_Label = new JLabel();
driverSGBD_Label.setBounds(new Rectangle(69, 165, 114, 16));
driverSGBD_Label.setText("Driver");

// ajout des labels au jContentPane de la fenêtre
jContentPane.add(nomUtilisateur_Label, null);
jContentPane.add(motDePasse_Label, null);
jContentPane.add(serverBD_Label, null);
jContentPane.add(driverSGBD_Label, null);

// ajout des champs de saisie et du champ Password au jContentPane
// les méthodes getXXX sont définies plus loin
// elles se chargent de la création des champs
jContentPane.add(getNomUtilisateur_TextField(), null);
jContentPane.add(getMotDePasse_PasswordField(), null);
jContentPane.add(getServerBD_TextField(), null);
jContentPane.add(getDriverSGBD_TextField(), null);

// ajout des boutons au jContentPane
// les méthodes getXXX sont définies plus loin
jContentPane.add(getBtn_Connexion(), null);
jContentPane.add(getBtn_Quitter(), null);
jContentPane.add(titre_Label, null);
}
return jContentPane;
}

// les méthodes getXXX() des champs de saisie et du champ Password
// -----
private JTextField getNomUtilisateur_TextField() {
    if (nomUtilisateur_TextField == null) {
        nomUtilisateur_TextField = new JTextField();
        nomUtilisateur_TextField.setBounds(new Rectangle(185, 88, 180, 20));
    }
}
private JPasswordField getMotDePasse_PasswordField() {
    if (motDePasse_PasswordField == null) {
        motDePasse_PasswordField = new JPasswordField(20);
        motDePasse_PasswordField.setBounds(new Rectangle(185, 115, 180, 20));
    }
}
private JTextField getServerBD_TextField() {
    if (serverBD_TextField == null) {
        serverBD_TextField = new JTextField();
        serverBD_TextField.setBounds(new Rectangle(185, 139, 180, 20));
    }
    return serverBD_TextField;
}
private JTextField getDriverSGBD_TextField() {
    if (driverSGBD_TextField == null) {
        driverSGBD_TextField = new JTextField();
        driverSGBD_TextField.setBounds(new Rectangle(185, 164, 180, 20));
    }
    return driverSGBD_TextField;
}

// les méthodes getXXX() des boutons
// création et initialisation
private JButton getBtn_Connexion() {
    if (btn_Connexion == null) {
        btn_Connexion = new JButton();
        btn_Connexion.setBounds(new Rectangle(106, 213, 119, 35));
        btn_Connexion.setText("Se connecter");
        btn_Connexion.addKeyListener(new java.awt.event.KeyAdapter() {

```

```

        });
    }
    return btn_Connexion;
}

private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(224, 213, 119, 35));
        btn_Quitter.setText("Quitter");
    }
    return btn_Quitter;
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new FenConnexion();
        }
    });
}
}

```

Fenêtre Menu

1. Fonctionnalités

L'utilisateur est accueilli par une fenêtre après s'être connecté à partir de la fenêtre de connexion. Cette fenêtre nommée **FenMenu** lui propose l'accès aux données de son application. Il peut ensuite mener les actions classiques de consultation, d'ajout, de suppression, de modification et de recherche sur les tables Clients, Articles et Commandes. Par le bouton **Paramètres**, il peut aussi accéder et mettre à jour les paramètres de gestion tels que les taux de TVA, la monnaie par défaut, le format des dates, etc.



2. Création de la maquette

Nous procédons comme pour la fenêtre de connexion en établissant une maquette sans nous soucier des actions attendues suite au choix et à la validation d'un bouton.

- Ouvrez la classe **FenMenu** à partir du paquetage **dialogue** en effectuant un clic droit sur la classe et en choisissant **Ouvrir avec - Visual Editor**.
- Modifiez la propriété **layout** du panneau nommé **jContentPane** à **null**.
- Cliquez sur **Swing Components** dans la palette sur le côté droit.
- Ajoutez les composants en les nommant comme indiqué ci-après.
 - l'étiquette pour le titre : **titre_Label** ;
 - les boutons : **btn_Clients**, **btn_Articles**, **btn_Commandes**, **btn_Parametres**, **btn_Quitter** ;
- N'oubliez pas d'importer la classe **ImageIcon**.

Rien de particulier au niveau du code par rapport à la construction de la fenêtre de connexion vue précédemment.

Notre étude étant circonscrite à la gestion des clients, seuls les boutons **CLIENTS** et **QUITTER** seront activés. Dans la section suivante, nous allons aborder (enfin) la gestion des événements qui donnera vie à nos premières fenêtres : **FenConnexion** et **FenMenu**.

Code complet de la classe FenMenu version 1 :

```
package dialogue;

import javax.swing.JFrame;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.JButton;

import java.awt.Rectangle;
import java.awt.Font;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class FenMenu extends JFrame {

    // PROPRIETES

    private static final long serialVersionUID = 1L;

    private JPanel jContentPane = null;
    private JButton btn_Clients = null;
    private JButton btn_Articles = null;
    private JLabel titre_Label = null;
    private JButton btn_Commandes = null;
    private JButton btn_Parametres = null;
    private JButton btn_Quitter = null;

    // CONSTRUCTEUR
    public FenMenu() {
        super();
        initialize();
    }

    // METHODES
    private void initialize() {
        this.setSize(361, 401);
        this.setContentPane(getJContentPane());
        this.setTitle("HL");
        ImageIcon image = new ImageIcon("images\\article.gif");
        this.setIconImage(image.getImage());
        this.setLocationRelativeTo(null);
    }

    private JPanel getJContentPane() {
        if (jContentPane == null) {
            titre_Label = new JLabel();
            titre_Label.setBounds(new Rectangle(6, 9, 342, 42));
            titre_Label.setHorizontalAlignment(SwingConstants.CENTER);
            titre_Label.setHorizontalTextPosition(SwingConstants.CENTER);
            titre_Label.setText("SARL CAGOU");
            titre_Label.setFont(new Font("Comic Sans MS", Font.BOLD, 24));
            jContentPane = new JPanel();
            jContentPane.setLayout(null);
            jContentPane.add(getBtn_Clients(), null);
            jContentPane.add(getBtn_Articles(), null);
            jContentPane.add(titre_Label, null);
            jContentPane.add(getBtn_Commandes(), null);
            jContentPane.add(getBtn_Parametres(), null);
            jContentPane.add(getBtn_Quitter(), null);
            jContentPane.setLayout(null);
        }
        return jContentPane ;
    }
}
```

```

}
private JButton getBtn_Clients() {
    if (btn_Clients == null) {
        btn_Clients = new JButton();
        btn_Clients.setBounds(new Rectangle(101, 74, 160, 51));
        btn_Clients.setText("CLIENTS");
        btn_Clients.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Clients.setIcon(new ImageIcon("images\\client.gif"));
    }
    return btn_Clients;
}
private JButton getBtn_Articles() {
    if (btn_Articles == null) {
        btn_Articles = new JButton();
        btn_Articles.setBounds(new Rectangle(101, 122, 160, 51));
        btn_Articles.setText("ARTICLES");
        btn_Articles.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Articles.setIcon(new ImageIcon("images\\article.gif"));
    }
    return btn_Articles;
}
private JButton getBtn_Commandes() {
    if (btn_Commandes == null) {
        btn_Commandes = new JButton();
        btn_Commandes.setBounds(new Rectangle(101, 170, 160, 51));
        btn_Commandes.setIcon(new ImageIcon("images\\commande.gif"));
        btn_Commandes.setText("COMMANDES");
        btn_Commandes.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
    }
    return btn_Commandes;
}
private JButton getBtn_Parametres() {
    if (btn_Parametres == null) {
        btn_Parametres = new JButton();
        btn_Parametres.setBounds(new Rectangle(101, 218, 160, 51));
        btn_Parametres.setIcon(new ImageIcon("images\\parametre.gif"));
        btn_Parametres.setText("PARAMETRES");
        btn_Parametres.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
    }
    return btn_Parametres;
}
private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(101, 266, 160, 51));
        btn_Quitter.setText("QUITTER");
        btn_Quitter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Quitter.setIcon(new ImageIcon("images\\Earth.gif"));
    }
    return btn_Quitter;
}
}
}

```

Gestion des événements

Nous avons créé précédemment les maquettes des fenêtres **FenConnexion** et **FenMenu**. Dans cette section, nous allons voir comment les rendre en partie opérationnelles en prenant en compte les actions que l'utilisateur peut mener à partir du clavier ou de la souris. Il nous faut auparavant approfondir le concept d'événement abordé au chapitre Concepts de base de la POO.

1. Notion avancée d'événement

Nous allons voir plus en détail la mise en œuvre des événements et leur traitement avec Java.

En Java, hormis les primitives, tout est objet. Les événements eux-mêmes sont des objets. Plus précisément, ce sont des instances de classes dont les noms se terminent par **Event**, par exemple : **ActionEvent**, **FocusEvent**, **HyperlinkEvent**, **MouseEvent**, etc. Java prend en compte de nombreux événements qui sont répartis pour des raisons historiques dans les paquetages **java.awt.event** et **javax.swing.event**.

Mais qui génère l'événement en tant qu'objet ? Autrement dit, quelle est la source qui crée l'objet événement ? Nous pensons de suite à l'utilisateur qui effectue un clic de souris ou appuie sur une touche (mais il pourrait aussi s'agir d'un autre objet). S'il est effectivement à l'origine de l'événement, l'utilisateur ne peut et cela va de soi, créer lui-même l'objet événement au sein du programme Java. C'est en fait le composant lui-même qui réalise cette création suite à une sollicitation externe et qui envoie ensuite l'objet événement à l'écouteur.

Tous les événements ne peuvent convenir à tous les composants. Par exemple, l'événement **WindowEvent** (ouverture, fermeture...) concerne le composant **Window** et ses dérivés (**JFrame**, **JDialog**) mais n'est de toute évidence d'aucune utilité pour le composant **JButton**. Par contre, celui-ci est concerné avec d'autres composants tels que **JMenuItem** ou **JList** par l'événement **ActionEvent** (activation d'un bouton, choix dans un menu ou dans une liste).

Pour chaque type d'événement, il existe donc un type d'écouteur (au moins un et le plus souvent plusieurs). Celui-ci est facilement identifiable. Par exemple, aux événements **ActionEvent** et **WindowEvent** correspondent les listeners (écouteurs) **ActionListener** et **WindowListener**.

Tout comme les événements, les listeners sont des instances mais de classes qui implémentent des interfaces particulières de type **Listener**. Ainsi, aux événements issus des classes **ActionEvent** et **WindowEvent**, sont associés les listeners issus de classes qui implémentent les interfaces **ActionListener** et **WindowListener**.

Il reste à ajouter les listeners aux composants, opération aisée ceux-ci possédant à leur création les méthodes adéquates du type **addXXXListener()**. Exemple : **addActionListener()** pour **ActionEvent** et **addWindowListener()** pour **WindowListener**.

Passons maintenant aux traitements des événements. C'est un peu plus compliqué car ils peuvent être réalisés de diverses manières. Nous pouvons procéder comme mentionné plus haut avec la création d'une classe implémentant l'interface retenue mais cela nécessite de redéfinir toutes ses méthodes (toutes n'étant pas forcément utiles).

Exemple avec l'interface **WindowListener** qui possède sept méthodes :

```
package packageCagou
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
...

// implémentation de l'interface WindowListener
public class Fenetre extends JFrame implements WindowListener{
...
    // constructeur
    public Fenetre() {
        super();
        initialize();
    }

    private void initialize() {
...
        // ajout du listener à la fenêtre
        this.addWindowListener(this);
    }
...
    /*
     * redéfinition de toutes les méthodes de l'interface WindowListener
     * par une définition vide
     * sauf ici pour la méthode windowClosing()
     */
    public void windowActivated(WindowEvent arg0) {}
    public void windowClosed(WindowEvent arg0) {}
    public void windowDeactivated(WindowEvent arg0) {}
    public void windowDeiconified(WindowEvent arg0) {}
    public void windowIconified(WindowEvent arg0) {}
    public void windowOpened(WindowEvent arg0) {}
    public void windowClosing(WindowEvent arg0) {
        System.out.println("La fenêtre a été fermée");
    }

    public static void main(String[] args) {
...
        new Fenetre();
    }
}
```

Pour éviter de nombreuses redéfinitions inutiles, il faut recourir aux **adaptateurs**. Ce sont des classes particulières du type **XXXAdapter** qui ont déjà redéfini pour nous avec une définition vide toutes les méthodes des interfaces **XXXListener**. Ainsi, aux interfaces **ActionListener** et **WindowListener** correspondent les adaptateurs **ActionAdapter** et **WindowAdapter**. Si nous reprenons l'exemple précédent, nous n'avons plus qu'une seule méthode à redéfinir :

```

package packageCagou
import java.awt.event.WindowAdapter;
...

/* l'implémentation de l'interface WindowsListener est à retirer
   car nous utilisons à sa place l'adaptateur WindowAdapter qui est une classe
public class Fenetre extends JFrame {
...
    // constructeur
    public Fenetre() {
        super();
        initialize();
    }

    private void initialize() {
...
        // ajout à la fenêtre de l'adaptateur en tant que listener
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.out.println("La fenêtre a été fermée");
            }
        });
    }
    /* La redéfinition de toutes les autres méthodes de l'interface WindowsListener
       n'a plus lieu d'être. Elle a donc été supprimée. */
}
public static void main(String[] args) {
...
    new Fenetre();
...
}
}

```

D'autres manières de traiter les événements sont possibles. Nous aurions pu par exemple définir au sein de la classe fenêtre, des classes internes étendant **WindowsListener** ou **WindowsAdapter** puis procéder à leur enregistrement auprès de la fenêtre.

Pour finir cette introduction aux événements, retenons que :

- les composants sont les sources des événements ;
- les événements sont détectés par les écouteurs ;
- les traitements sont réalisés au sein des écouteurs.

Passons maintenant à la mise en œuvre avec Eclipse.

2. Mise en œuvre

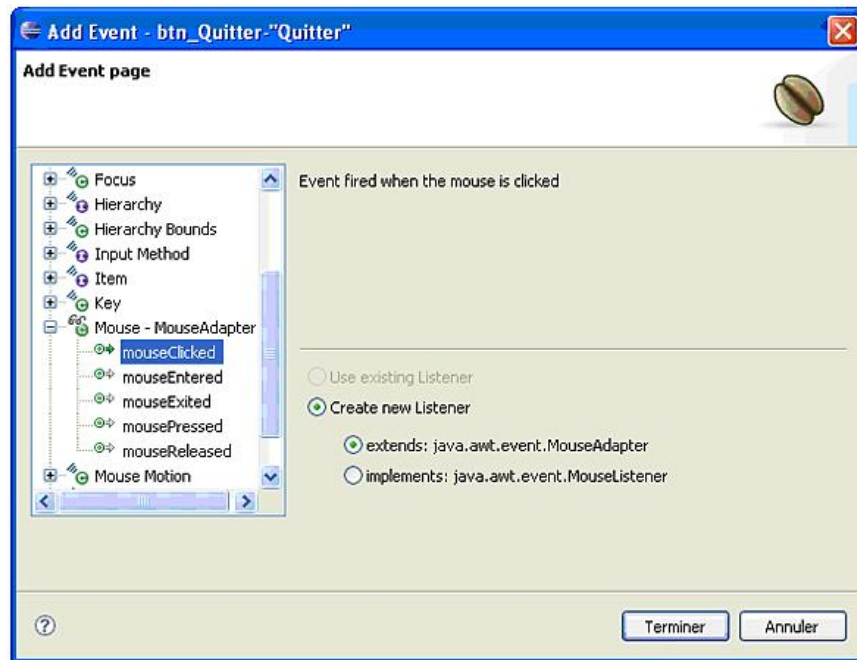
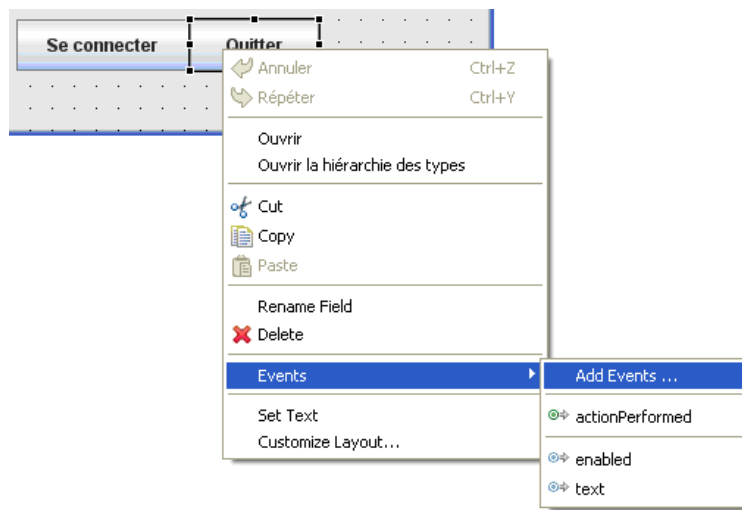
- Dans l'explorateur de paquetages, effectuez un clic droit sur la classe **FenConnexion**. Choisissez **Ouvrir avec - Visual Editor**.

Nous allons commencer par le bouton **Quitter** qui est plus le simple à gérer. Deux types d'événements peuvent le concerner : clic de souris mais aussi la pression sur la touche [Entrée] pour gagner du temps.

a. Gestion de la souris

Commençons par gérer l'événement clic de souris.

- Effectuez un clic droit sur le bouton **Quitter** et procédez aux choix ci-après.



Dans la zone de code, vous pouvez constater l'insertion des lignes correspondant à votre choix.

```
private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(224, 213, 119, 35));
        btn_Quitter.setText("Quitter");
        btn_Quitter.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                System.out.println("mouseClicked()");
            }
        });
    }
    return btn_Quitter;
}
```

L'ajout de l'écouteur adéquat au bouton par le biais de l'adaptateur **MouseListener** a été effectué et la méthode **mouseClicked()** a été redéfinie.

Nous pouvons constater qu'Eclipse réalise par défaut l'importation du paquetage **java.awt.event** au moment de l'instanciation de l'adaptateur **MouseListener**.

- Remplacez l'instruction **System.out.println(...)** par :

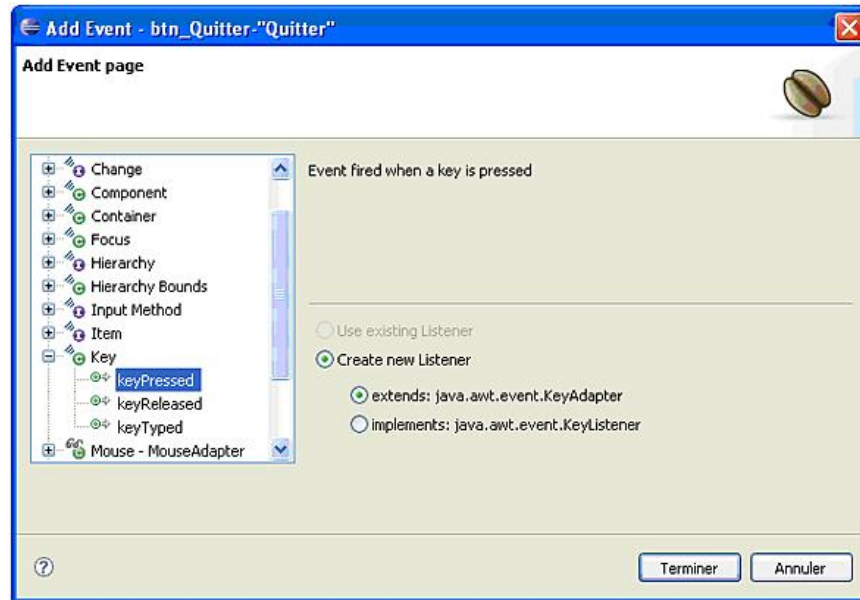
```
FenConnexion.this.dispose();
System.exit(0);
```

- Lancez la fenêtre et testez le bouton **Quitter** avec la souris.

b. Gestion du clavier

Traitions maintenant le deuxième type d'événement : appui sur la touche [Entrée].

- Effectuez à nouveau un clic droit sur le bouton **Quitter** et procédez aux choix ci-après.



Nous obtenons les lignes de code suivantes :

```
btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {  
    public void keyPressed(java.awt.event.KeyEvent e) {  
        System.out.println("keyPressed()");  
    }  
});
```

- Remplacez à nouveau l'instruction **System.out.println(...)** par :

```
FenConnexion.this.dispose();  
System.exit(0);
```

- Lancez la fenêtre et testez le bouton **Quitter** avec la touche [Entrée] puis avec des touches quelconques du clavier.

Vous constatez que n'importe quelle touche pressée provoque la fin de l'application. Bien sûr, nous désirons qu'une seule touche permette cette action, en l'occurrence la touche [Entrée] quand le bouton **Quitter** a le focus. Java met à notre disposition la méthode **getKeyCode()** qui permet d'obtenir le code ASCII de la touche pressée. Nous pouvons donc facilement programmer une action propre à cet événement.



Un composant a le focus quand il est sélectionné soit par la souris soit par l'utilisation du clavier, le plus souvent avec la touche [Tab]. Tant qu'il dispose du focus, il est le seul composant concerné par une action de la souris ou du clavier et réagit s'il a été programmé pour détecter les événements générés par ces derniers.

- Effectuez les modifications suivantes puis testez à nouveau le bouton **Quitter** avec le clavier.

```
btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {public void keyPressed(java.awt.event.KeyEvent e) {if(e.getKeyCode()== 10){  
    FenConnexion.this.dispose();System.exit(0);  
}  
}});
```

Nous allons maintenant traiter le bouton **Se connecter**. Tout comme le bouton **Quitter**, nous prenons en compte les événements de type clic de souris et appui sur la touche [Entrée].

- Procédez comme vu précédemment. Le code obtenu est le suivant :

```
private JButton getBtn_Connexion() {  
    if (btn_Connexion == null) {  
        btn_Connexion = new JButton();  
        btn_Connexion.setBounds(new Rectangle(106, 213, 119, 35));  
        btn_Connexion.setText("Se connecter");  
        btn_Connexion.addMouseListener(new java.awt.event.MouseAdapter() {  
            public void mouseClicked(java.awt.event.MouseEvent e) {  
                System.out.println("mouseClicked()");  
            }  
        })  
    }  
}
```

```

    });
    btn_Connexion.addKeyListener(new java.awt.event.KeyAdapter() {
        public void keyPressed(java.awt.event.KeyEvent e) {
            System.out.println("keyPressed()");
        }
    });
}
return btn_Connexion;
}

```

- Lancez la fenêtre et testez le bouton **Se connecter**.

Tout est désormais prêt pour réaliser l'enchaînement sur la fenêtre **FenMenu**. Nous allons remplacer les messages apparaissant en mode console par l'ouverture de la fenêtre désirée.

- Modifiez le code.

```

private JButton getBtn_Connexion() {
    ...
    public void mouseClicked(java.awt.event.MouseEvent e) {
        FenConnexion.this.dispose();
        FenMenu laFenetreMenu = new FenMenu();
        laFenetreMenu.setVisible(true);
    }
    ...
    public void keyPressed(java.awt.event.KeyEvent e) {
        // filtre pour la touche Entrée
        if(e.getKeyCode() == 10){
            FenConnexion.this.dispose();
            FenMenu laFenetreMenu = new FenMenu();
            laFenetreMenu.setVisible(true);
        }
    }
    ...
}

```

Pour la maquette **FenMenu**, selon les mêmes principes, les boutons **CLIENTS** et **QUITTER** sont rendus opérationnels.

- Dans l'explorateur de paquetages, effectuez un clic droit sur la classe **FenMenu**. Choisissez **Ouvrir avec - Visual Editor**.
- Modifiez le code comme vu précédemment en l'adaptant au bouton.

```

private JButton getBtn_Clients() {
    if (btn_Clients == null) {
        btn_Clients = new JButton();
        btn_Clients.setBounds(new Rectangle(101, 74, 160, 51));
        btn_Clients.setText("CLIENTS");
        btn_Clients.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Clients.setIcon(new ImageIcon("images\\client.gif"));
        btn_Clients.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                FenTableClient laFenetre_TableClient = new FenTableClient();
                laFenetre_TableClient.setVisible(true);
            }
        });
        btn_Clients.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(java.awt.event.KeyEvent e) {
                if(e.getKeyCode() == 10){
                    FenTableClient laFenetre_TableClient = new FenTableClient();
                    laFenetre_TableClient.setVisible(true);
                }
            }
        });
    }
    return btn_Clients;
}

private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(101, 266, 160, 51));
        btn_Quitter.setText("QUITTER");
        btn_Quitter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Quitter.setIcon(new ImageIcon("images\\Earth.gif"));
        btn_Quitter.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                FenMenu.this.dispose();
                System.exit(0);
            }
        });
        btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {

```

```

        public void keyPressed(java.awt.event.KeyEvent e) {
            if(e.getKeyCode() == 10){
                FenMenu.this.dispose();
                System.exit(0);
            }
        }
    }
};
}
return btn_Quitter;
}

```

Nous ne reviendrons plus sur le code de la fenêtre **FenMenu**. Par contre, il reste beaucoup à faire pour rendre la fenêtre **FenConnexion** totalement opérationnelle. C'est ce que nous allons traiter dans la section suivante.

Code de la classe FenConnexion version 2

La structure générale est juste rappelée, sauf pour les méthodes **getBtn_Connexion()** et **getBtn_Quitter()** dont le code est intégralement reporté.

```

// FenConnection Version 2

package dialogue;

// IMPORTATIONS
import javax.swing.JPanel;
...

public class FenConnexion extends dialogue.FenMenu {
    // PROPRIETES
    private static final long serialVersionUID = 1L;
    private JPanel jContentPane ;
    ...

    // CONSTRUCTEUR
    public FenConnexion() {
    ...
    }

    // METHODES
    private void initialize() {
    ...

    private JButton getBtn_Connexion() {
        if (btn_Connexion == null) {
            btn_Connexion = new JButton();
            btn_Connexion.setBounds(new Rectangle(106, 213, 119, 35));
            btn_Connexion.setText("Se connecter");
            btn_Connexion.addMouseListener(new java.awt.event.MouseAdapter() {
                public void mouseClicked(java.awt.event.MouseEvent e) {
                    FenConnexion.this.dispose();
                    FenMenu laFenetreMenu = new FenMenu();
                    laFenetreMenu.setVisible(true);
                }
            });
            btn_Connexion.addKeyListener(new java.awt.event.KeyAdapter() {
                public void keyPressed(java.awt.event.KeyEvent e) {
                    if(e.getKeyCode() == 10){
                        FenConnexion.this.dispose();
                        FenMenu laFenetreMenu = new FenMenu();
                        laFenetreMenu.setVisible(true);
                    }
                }
            });
        }
        return btn_Connexion;
    }

    private JButton getBtn_Quitter() {
        if (btn_Quitter == null) {
            btn_Quitter = new JButton();
            btn_Quitter.setBounds(new Rectangle(224, 213, 119, 35));
            btn_Quitter.setText("Quitter");
            btn_Quitter.addMouseListener(new java.awt.event.MouseAdapter() {
                public void mouseClicked(java.awt.event.MouseEvent e) {
                    FenConnexion.this.dispose();
                    System.exit(0);
                }
            });
            btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {
                public void keyPressed(java.awt.event.KeyEvent e) {
                    if(e.getKeyCode() == 10){
                        FenConnexion.this.dispose();
                        System.exit(0);
                    }
                }
            });
        }
    }
}

```

```
        return btn_Quitter;
    }

    public static void main(String[] args) {
        ...
    }
}
```

Gestion de la connexion


Faisons le point : les maquettes de fenêtres de connexion et de menu ont été créées et la gestion des événements a été traitée dans la section précédente. Il s'agit dans cette section de mettre en place tous les contrôles nécessaires avant d'autoriser l'accès à l'application. Simultanément, il faut gérer l'accès au serveur et à la base de données en prenant en compte les problèmes possibles de connexion.


1. Paramètres de connexion

Nous allons ajouter à la classe **Parametres** toutes les informations utiles à la connexion :

- nom ;
 - mot de passe ;
 - pont JDBC/ODBC ;
 - source de données ODBC pour l'accès à la base de données.
- Ouvrez la classe **Parametres** à partir du paquetage **entite** avec l'éditeur Java.
 - Ajoutez les attributs et le constructeur.

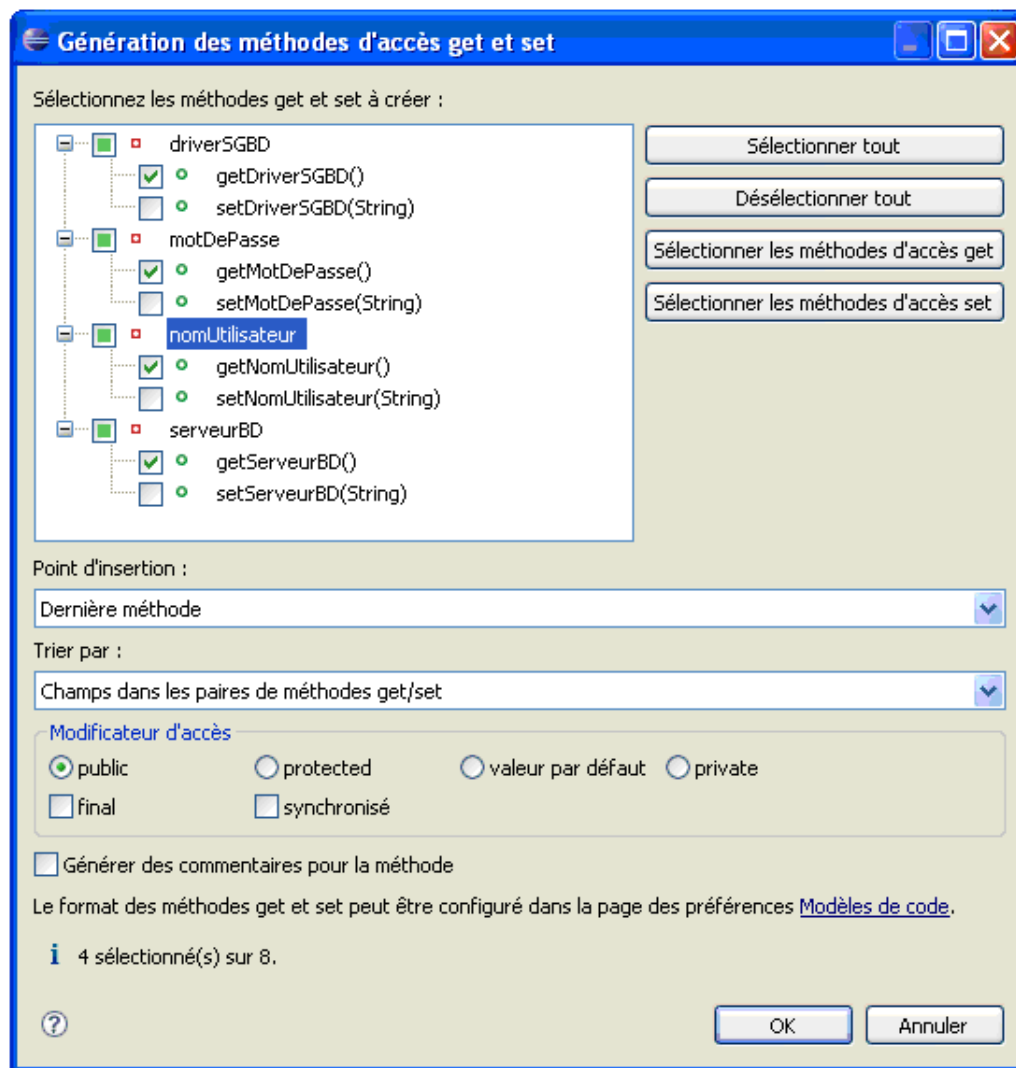
```
private String nomUtilisateur;  
private String motDePasse;  
private String serveurBD;  
private String driverSGBD;  
  
// Constructeur  
public Parametres () {  
    nomUtilisateur = "root";  
    motDePasse = "tempo";  
    driverSGBD = "sun.jdbc.odbc.JdbcOdbcDriver";  
    serveurBD = "jdbc:odbc:BDJavaCagou";  
}
```

 Pour faire simple, le nom de l'utilisateur et le mot de passe sont uniques. Les paramètres concernant le pilote, le serveur et la base de données proviennent de l'installation réalisée pour le SGBDR MySQL (cf. chapitre Base de données MySQL).

 Pour une exploitation réelle, ne jamais mettre en clair le mot de passe. Une solution possible consiste à le crypter avec un script PHP en utilisant la méthode md5() puis à le stocker dans la base de données MySQL.


Passons à l'ajout des accesseurs. Cette opération est possible et rapide avec Eclipse.


- Sélectionnez un attribut dans l'éditeur de code et effectuez un clic droit.
- Choisissez les options **Source** puis **Générer les méthodes d'accès get et set**.
- Cochez seulement les accesseurs (méthodes getXXX) et validez.



Le code est généré.

```
public String getDriverSGBD() {
    return driverSGBD;
}
public String getMotDePasse() {
    return motDePasse;
}
public String getNomUtilisateur() {
    return nomUtilisateur;
}
public String getServeurBD() {
    return serveurBD;
}
```

 Afin de respecter le principe d'encapsulation, les attributs sont déclarés **private** et ne sont accessibles que par les accesseurs qui eux sont déclarés **public**.

 Les mutateurs (méthodes setXXX) ne sont pas utilisés ici. Il est possible de les ajouter afin de pouvoir modifier les paramètres par défaut. Ceci implique une gestion plus fine des utilisateurs. Seuls l'administrateur ou le concepteur devraient alors pouvoir effectuer les modifications concernant le driver et la base de données.

D'autres écritures de cette classe sont possibles :

- la déclarer **abstract** : elle ne peut être instanciée. L'accès aux membres est réalisé selon la syntaxe classe.accesseur et s'effectue directement sans recherche dynamique.

- déclarer ses attributs statiques et les initialiser : elle peut être instanciée mais les instances de la classe ne possèdent pas les attributs. L'accès peut être réalisé de la même manière ou selon la syntaxe objet.accesseur, l'accès aux attributs se faisant alors de manière dynamique.

2. Connexion au serveur et à la base de données

La classe **ControleConnexion** aborde en particulier les points suivants :

- importation des classes de l'API JDBC ;
- gestion des exceptions ;
- les méthodes statiques.

Elle comporte les méthodes assurant la connexion :

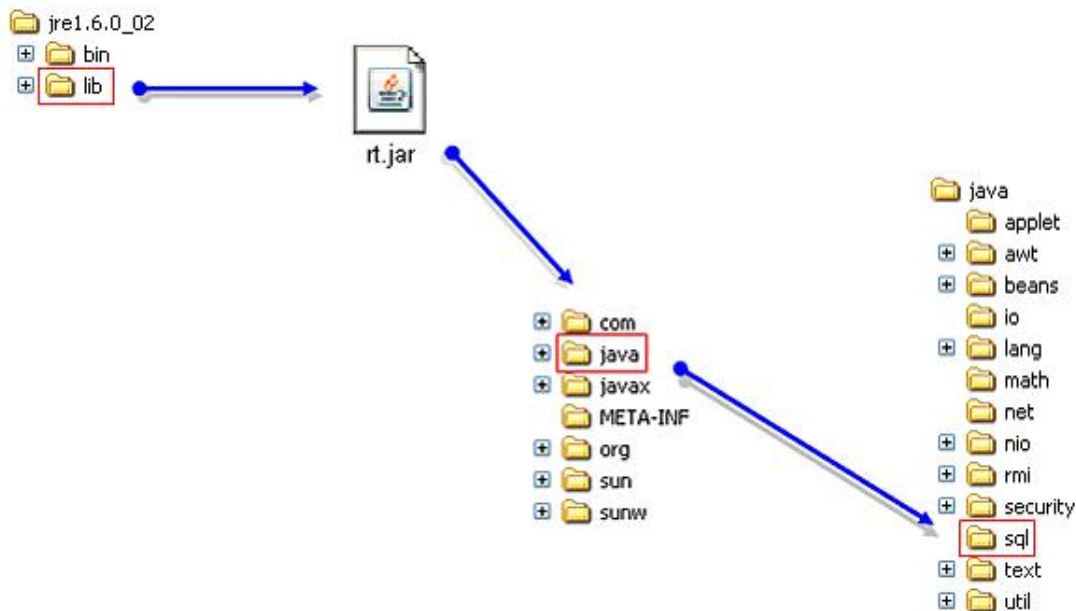
- chargement et enregistrement du driver MySQL ;
 - connexion et authentification.
- Ouvrez la classe **ControleConnexion** à partir du paquetage **controle** avec l'éditeur Java et complétez le code. Commencez par les importations :
- pour la connexion ;
 - pour établir le lien avec la classe **Parametres** ;
 - pour l'affichage des boîtes de dialogue.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import entite.Parametres;


import javax.swing.JOptionPane;
```

Nous avons procédé à une importation sélective des classes nécessaires à l'utilisation de JDBC présentes dans le paquetage java.sql.



Passons aux propriétés. Le lien avec la classe **Parametres** est assuré par une propriété. Une autre propriété conserve l'état de connexion.

```
static Parametres lesParametres;
static boolean etatConnexion;
```

 Les propriétés statiques appartiennent à la classe et non aux objets de la classe.


L'intérêt d'une connexion statique est de disposer d'une connexion unique utilisable durant toute une session. L'initialisation s'effectue immédiatement après la déclaration avec un initialiseur statique.

```
static Connection laConnexionStatique;
static {
}
```

Lors de l'initialisation plusieurs tâches sont réalisées. En premier, l'enregistrement du pilote ODBC auprès du gestionnaire de pilotes et son chargement puis la connexion au serveur et à la base de données.

L'enregistrement du pilote s'effectue avec le code suivant :

```
Class.forName(lesParametres.getDriverSGBD());
```

 **Class** correspond à la classe du pilote. Avec sa méthode statique **forName()**, elle crée une instance d'elle-même - qui est donc chargée en mémoire - et l'enregistre auprès de la classe **DriverManager**.

L'insertion de cette ligne de code provoque cependant une erreur dûment signalée par Eclipse.



Le mot clé **try** est toujours suivi d'un bloc d'instructions qui définit le traitement à exécuter. Si celui-ci devait échouer,

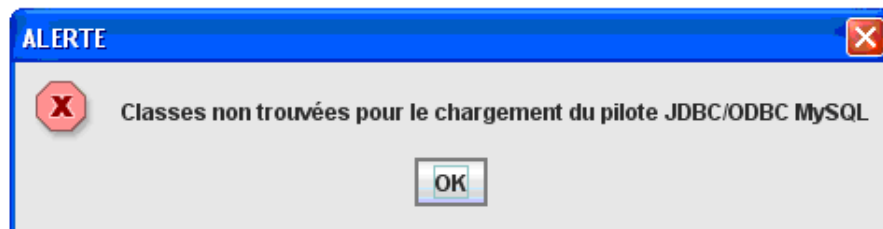
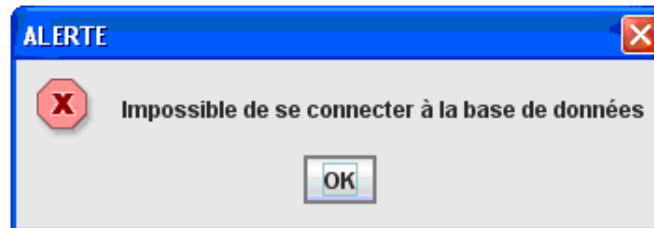
une erreur ou **exception** est créée ou "levée". Dans ce cas, l'exception est prise en considération ou "attrapée" et traitée dans le bloc du **catch** qui doit toujours être placé immédiatement après le bloc **try**.

Tout traitement concernant les bases de données doit être géré en tenant compte des nombreux problèmes possibles. Java impose que ceux-ci soient pris en compte et Eclipse nous aide en proposant de les entourer d'un bloc **try/catch**.

Des variantes de la gestion des exceptions existent. Par exemple, on peut indiquer qu'un traitement est susceptible de lancer une exception en utilisant le mot clé **throws exception**. Pour de plus amples informations sur la gestion des exceptions, consulter l'aide de Sun.

```
java.lang.Object
└─ java.lang.Throwable
    └─ java.lang.Exception
```

La gestion des exceptions est faite ici en deux temps. On se soucie d'abord du pilote pour le SGBDR MySQL. Si aucune exception n'est rencontrée, une variable booléenne indique que le deuxième traitement peut être effectué. Cette structure permet de proposer des messages personnalisés en fonction de l'exception levée.



```
static Connection laConnectionStatique;static {
    boolean ok = true;
    lesParametres = new Parametres();

    // 1. Enregistrement du pilote ODBC
    // -----
    try {
        Class.forName(lesParametres.getDriverSGBD());
        etatConnexion = true;
    }
    catch(ClassNotFoundException e){
        JOptionPane.showMessageDialog(null, "Classes non trouvées"
        + " pour le chargement du pilote JDBC/ODBC MySQL",
        "ALERTE", JOptionPane.ERROR_MESSAGE);
        ok = false;
        etatConnexion = false;
    }

    // 2. Etablissement de la connexion
    // -----
    if (ok == true){
        try {
            // récupération des paramètres présents dans la classe Parametres
            String urlBD = lesParametres.getServeurBD();
            String nomUtilisateur = lesParametres.getNomUtilisateur();
            String MDP = lesParametres.getMotDePasse();
            // Création d'une connexion contenant les paramètres de connexion
            laConnectionStatique = DriverManager.getConnection(urlBD, nomUtilisateur, MDP);
            etatConnexion = true;
        }
        catch (Exception e) {
```

```

JOptionPane.showMessageDialog(null, "Impossible de se connecter" +
" à la base de données",
"ALERTE", JOptionPane.ERROR_MESSAGE);
etatConnexion = false;
}
}
}

```



Une instance de la classe **Parametres** est créée pour récupérer les paramètres.

La méthode statique **forName()** de la classe **Class** peut lever une exception de la classe **ClassNotFoundException** présente dans le paquetage `java.lang`.

L'attribut statique booléen **etatConnexion** est initialisé à la sortie de la méthode.

Ajout du constructeur

```

public ControleConnexion(){
}

```

Il reste à ajouter les accesseurs. Ils sont tous statiques car ils correspondent à des méthodes de portée classe. En outre, il n'est pas possible d'établir une référence à une propriété statique sans passer par une méthode statique.

```

public static Parametres getParametres(){
    return lesParametres;
}

public static boolean getControleConnexion(){
    return etatConnexion;
}

public static Connection getConnexion(){
    return laConnexionStatique;
}
}

```

La connexion n'est pas pour autant réalisée. Les paramètres de l'utilisateur doivent correspondre aux valeurs des propriétés correspondantes de la classe **Parametres**. Un contrôle est nécessaire.

```

public static boolean controle(String Nom, String MotDePasse){
    boolean verificationSaisie;
    if (Nom.equals(lesParametres.getNomUtilisateur())
    && MotDePasse.equals(lesParametres.getMotDePasse())){
        verificationSaisie = true;
    }
    else {
        JOptionPane.showMessageDialog(null, "Vérifier votre saisie.",
        "ERREUR", JOptionPane.ERROR_MESSAGE);
        verificationSaisie = false;
    }
    return verificationSaisie;
}

```

Il reste à doter la classe **ControleConnexion** d'une méthode pour fermer la connexion à la fin de l'exécution de l'application.

```

public static void fermetureSession(){
    try {
        laConnexionStatique.close();
    }
    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Problème rencontré" +
        "à la fermeture de la connexion",
        "ERREUR", JOptionPane.ERROR_MESSAGE);
    }
}

```



La structure du code proposé permet de modifier au besoin les paramètres de connexion sans remettre en question les méthodes de connexion. Une suite possible à la gestion de la connexion :

- gestion de plusieurs catégories d'utilisateurs avec des droits différents dont les données sont stockées dans une base de données MySQL ;
- ajout d'un bouton dans la fenêtre de connexion accessible uniquement par l'administrateur ou le concepteur permettant de modifier le pilote en fonction du SGBD et le chemin d'accès à la base de données.

3. Activation de la fenêtre de connexion

La fenêtre **FenConnexion** va passer ici du statut de maquette à celui de fenêtre opérationnelle.

- Ouvrez la classe graphique **FenConnexion** avec Visual Editor. Nous allons compléter la méthode **getBtn_Connexion()** du bouton **btn_Connexion** (intitulé : **Se connecter**).

La classe **FenConnexion** est une IHM. Elle se contente de transmettre la demande de l'utilisateur à la classe **ControleConnection** pour la connexion à la base de données.

- Commencez par importer la classe **ControleConnection**..

```
...
import controle.ControleConnexion;
```

Nous allons ajouter un double contrôle, de la saisie et de la connexion, à la méthode **getBtn_Connexion()**. L'utilisateur est averti d'un éventuel problème de connexion. Si la saisie est cependant correcte, un message plus complet est rajouté à partir de la classe **FenConnexion** et vient compléter ceux déjà fournis par la classe **ControleConnection**.



```
private JButton getBtn_Connexion() {
    if (btn_Connexion == null) {
        ...
        btn_Connexion.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                String leNom = nomUtilisateur_TextField.getText();
                String leMotDePasse = String.valueOf(motDePasse_PasswordField.getPassword());
                if(ControleConnexion.controle(leNom, leMotDePasse)){
                    if(ControleConnexion.getControleConnexion()){
                        FenConnexion.this.dispose();
                        FenMenu laFenetreMenu = new FenMenu();
                        laFenetreMenu.setVisible(true);
                    }
                }
                else
                    JOptionPane.showMessageDialog(null, "Impossible de se connecter" +
                        " à la base de données" + '\n' + '\n'
                        + "Vos nom et mot de passe sont corrects." + '\n'
                        + "Mais les paramètres pour le pilote et la base de données "
                        + "doivent être vérifiés" + '\n' + '\n'
                        + "Conctacter le responsable informatique.",
                        "ALERTE", JOptionPane.ERROR_MESSAGE);
            }
        });
    }
}
```

```

    }
    }
    });
}
return btn_Connexion;
}

```



Le contrôle de saisie s'effectue avec la méthode statique **controle()** de la classe **ControleConnexion**.

L'expression **motDePasse_PasswordField.getPassword()** renvoie un tableau de caractères. Java fait une distinction entre les chaînes de caractères déclarées avec la classe **String** et les tableaux de caractères, exemple :

```

// accepté
char[] vChar;
vChar = motDePasse_PasswordField.getPassword();
// refusé
String vChaine;
vChaine = motDePasse_PasswordField.getPassword();

```

Il faut convertir le tableau de caractères en chaîne de caractères, conversion réalisée avec la méthode **valueOf()** de la classe **String**.

```
vPassword = String.valueOf(motDePasse_PasswordField.getPassword());
```

Pour incorporer des caractères spéciaux, il faut utiliser les caractères d'échappement placés entre des apostrophes (quotes) et constitués de l'antislash suivi d'une lettre.

Exemples :

- `\n` : nouvelle ligne.
- `\t` : tabulation.
- `\"` : guillemets.

- Testez la classe **FenConnexion** en modifiant les paramètres.



Vous pouvez affiner le contrôle de saisie en vérifiant si l'utilisateur a au moins renseigné les rubriques et en précisant si l'erreur vient du nom ou du mot de passe.

Il reste à effectuer les mêmes traitements pour le deuxième événement **mouseClicked()**. En référence à la programmation structurée, le plus simple est de créer une méthode commune aux deux événements. Avantages : moins de lignes de code et surtout une maintenance aisée.

La méthode **controleConnexion_Appel()** est créée rapidement par copier-coller.

```

private void controleConnexion_Appel(){
    String leNom = nomUtilisateur_TextField.getText();
    String leMotDePasse = String.valueOf(motDePasse_PasswordField.getPassword());
    ...
    "ALERTE", JOptionPane.ERROR_MESSAGE);
}
}

```

- Insérez cette méthode dans la méthode **getBtn_Connexion**.

```

private JButton getBtn_Connexion() {
    if (btn_Connexion == null) {
        ...
        btn_Connexion.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                controleConnexion_Appel();
            }
        });
    }
}

```

```

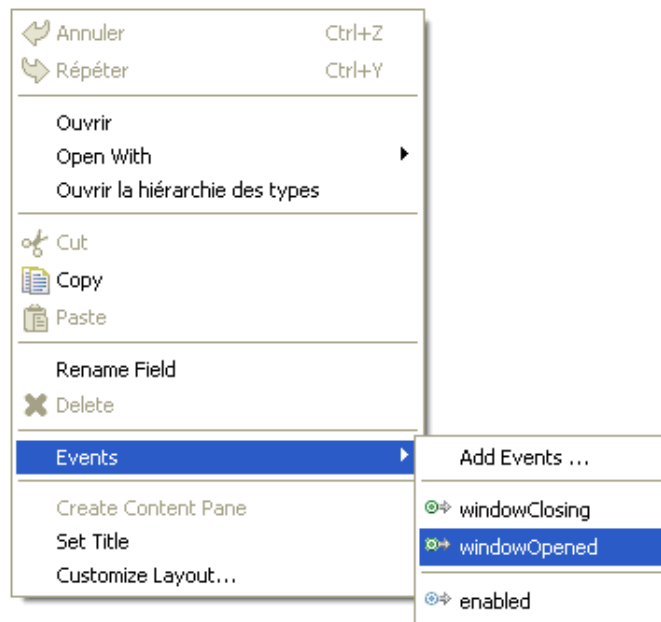
});

btn_Connexion.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent e) {if(e.getKeyCode()== 10){
        controleConnexion_Appel();
    }
});
}return btn_Connexion; }

```

Pour finir, les paramètres de connexion concernant le pilote et la base de données sont rendus visibles dans la fenêtre de connexion. Il s'agit d'une simple information, l'utilisateur ne pouvant modifier ces paramètres. Cela nous permet d'aborder la gestion des événements au niveau de la fenêtre.

- Pour être sûr de bien sélectionner la fenêtre et non le panneau, effectuez un clic droit sur l'un des coins de la fenêtre.
- Choisissez **Events - windowOpened**.




- Modifiez le code généré.

```

this.addWindowListener(new java.awt.event.WindowAdapter() {
    public void windowOpened(java.awt.event.WindowEvent e) {
        entite.Parametres leControleParametres = ControleConnexion.getParametres();
        serverBD_TextField.setText(leControleParametres.getDriverSGBD());
        driverSGBD_TextField.setText(leControleParametres.getServeurBD());
    }
});

```

 L'objet **leControleParametres** de type **Parametres** reçoit les paramètres récupérés par le biais de la méthode statique **getParametres()** de la classe **ControleConnexion**. L'importation de la classe **Parametres** n'ayant pas été effectuée, il faut faire précéder la classe du nom de son paquetage.

- La fenêtre de connexion est désormais totalement opérationnelle. Testez l'application. Vous devriez obtenir le résultat suivant :



Avant de clore cette section, nous allons modifier le bouton **QUITTER** de la fenêtre menu, **FenMenu**, en insérant le code qui ferme la connexion lorsque l'utilisateur quitte l'application.

- Ouvrez la classe **FenMenu** et effectuez les modifications.

```
private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        ...
        public void actionPerformed(java.awt.event.ActionEvent e) {
            FenMenu.this.dispose();
            controle.ControleConnexion.fermetureSession();
            System.exit(0);
        }
    };
    btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {
        public void keyPressed(java.awt.event.KeyEvent e) {
            if(e.getKeyCode() == 10){
                FenMenu.this.dispose();
                controle.ControleConnexion.fermetureSession();
                System.exit(0);
            }
        }
    });
}
return btn_Quitter;
}
```

Classe **Parametres** version finale :

```
package entite;

public class Parametres {

    // PROPRIETES
    private String nomUtilisateur;
    private String motDePasse;
    private String serveurBD;
    private String driverSGBD;

    // CONSTRUCTEUR
    public Parametres () {
        nomUtilisateur = "root";
        motDePasse = "tempo";
        driverSGBD = "sun.jdbc.odbc.JdbcOdbcDriver";
    }
}
```

```

    serveurBD = "jdbc:odbc:BDJavaCagou";
}

public String getDriverSGBD() {
    return driverSGBD;
}
public String getMotDePasse() {
    return motDePasse;
}
public String getNomUtilisateur() {
    return nomUtilisateur;
}
public String getServeurBD() {
    return serveurBD;
}
}

```

Classe **ControleConnexion** version finale :

```

package controle;

import entite.Parametres;

//importation des classes pour JDBC
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// pour les boîtes de dialogue
import javax.swing.JOptionPane;

public class ControleConnexion {

    // PROPRIETES
    static Parametres lesParametres;
    static boolean etatConnexion;
    static Connection laConnexionStatique;
    static {
        boolean ok = true;
        lesParametres = new Parametres();
        try {
            Class.forName(lesParametres.getDriverSGBD());
            etatConnexion = true;
        }
        catch(ClassNotFoundException e){
            JOptionPane.showMessageDialog(null, "Classes non trouvées"
                + " pour le chargement du pilote JDBC/ODBC MySQL",
                "ALERTE", JOptionPane.ERROR_MESSAGE);
            ok = false;
            etatConnexion = false;
        }
        if (ok == true){
            try {
                // récupération des paramètres présents dans la classe Parametres
                String urlBD = lesParametres.getServeurBD();
                String nomUtilisateur = lesParametres.getNomUtilisateur();
                String MDP = lesParametres.getMotDePasse();
                laConnexionStatique = DriverManager.getConnection(urlBD, nomUtilisateur, MDP);
                etatConnexion = true;
            }
            catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Impossible de se connecter" +
                    " à la base de données",
                    "ALERTE", JOptionPane.ERROR_MESSAGE);
                etatConnexion = false;
            }
        }
    }
}

```



```

    }
}

// CONSTRUCTEUR
public ControleConnexion(){
}

// METHODES

    // les accesseurs statiques
    // -----
    public static Parametres getParametres(){
        return lesParametres;
    }
    public static boolean getControleConnexion(){
        return etatConnexion;
    }
    public static Connection getConnexion(){
        return laConnectionStatique;
    }

    // les autres méthodes
    // -----
    public static boolean controle(String Nom, String MotDePasse){
        boolean verificationSaisie;
        if (Nom.equals(lesParametres.getNomUtilisateur())
        && MotDePasse.equals(lesParametres.getMotDePasse())){
            verificationSaisie = true;
        }
        else {
            JOptionPane.showMessageDialog(null, "Vérifier votre saisie.",
            "ERREUR", JOptionPane.ERROR_MESSAGE);
            verificationSaisie = false;
        }
        return verificationSaisie;
    }
    public static void fermetureSession(){
        try {
            laConnectionStatique.close();
        }
        catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Problème rencontré" +
            "à la fermeture de la connexion",
            "ERREUR", JOptionPane.ERROR_MESSAGE);
        }
    }
}
}

```

Classe **FenetreConnexion** version finale :

La structure générale est rappelée. Seules, les méthodes **getBtn_Connexion** et **getBtn_Quitter** sont reportées dans leur intégralité.

```

package dialogue;

...

import controle.ControleConnexion;

public class FenConnexion extends JFrame {

    // PROPRIETES
    ...

    // CONSTRUCTEUR
    ...

```

```

// METHODES
...

// les méthodes getXXX() des boutons
// création et initialisation
private JButton getBtn_Connexion() {
    if (btn_Connexion == null) {
        btn_Connexion = new JButton();
        btn_Connexion.setBounds(new Rectangle(106, 213, 119, 35));
        btn_Connexion.setText("Se connecter");

        btn_Connexion.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                /* ACTION
                 * ***** */
                controleConnexion_Appel();
            }
        });

        btn_Connexion.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(java.awt.event.KeyEvent e) {
                if(e.getKeyCode()== 10){
                    controleConnexion_Appel();
                }
            }
        });
    }
    return btn_Connexion;
}

private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(224, 213, 119, 35));
        btn_Quitter.setText("Quitter");
        btn_Quitter.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                FenConnexion.this.dispose();
                System.exit(0);
            }
        });
        btn_Quitter.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(java.awt.event.KeyEvent e) {
                if(e.getKeyCode()== 10){
                    FenConnexion.this.dispose();
                    System.exit(0);
                }
            }
        });
    }
    return btn_Quitter;
}

/* METHODE ACTION
***** */

private void controleConnexion_Appel(){
    ControleConnexion leControleConnexion = new ControleConnexion();
    serverBD_TextField.setText(leControleConnexion.getParametres().getDriverSGBD());
    driverSGBD_TextField.setText(leControleConnexion.getParametres().getServeurBD());

    String leNom = nomUtilisateur_TextField.getText();
    String leMotDePasse = String.valueOf(motDePasse_PasswordField.getPassword());
    leControleConnexion.controle(leNom, leMotDePasse);
    if(leControleConnexion.getControleSaisie()){
        if(leControleConnexion.getControleConnexion()){
            FenConnexion.this.dispose();

```

```

        FenMenu laFenetreMenu = new FenMenu();
        laFenetreMenu.setVisible(true);
    }
    else
        JOptionPane.showMessageDialog(null, "Impossible de se connecter" +
            " à la base de données" +'\n' +'\n'
            + "Vos nom et mot de passe sont corrects." +'\n'
            + "Mais les paramètres pour le pilote et la base de données "
            + "doivent être vérifiés" +'\n' +'\n'
            + "Conctacter le responsable informatique.",
            "ALERTE", JOptionPane.ERROR_MESSAGE);
    }
}

public static void main(String[] args) {
    ...
}
}

```

Fenêtres Clients

Deux types d’affichage sont proposés pour les enregistrements provenant de la base de données MySQL :

- en mode table ;
- en mode fiche.

1. Fonctionnalités

Fenêtre affichant les données en mode table :

HL

CLIENTS

Code	Nom	Type
BO1	BAUMER SA	Société
DR1	DROUX	Particulier
LA1	LANGLE	Artisan

AJOUTER

SUPPRIMER

MODIFIER

RECHERCHER

QUITTER

Les données apparaissent dans un tableau. Quatre boutons sont proposés permettant à l'utilisateur d'effectuer certains traitements sur les données. L'activation d'un bouton présente une autre fenêtre en mode fiche.

Fenêtre affichant les données en mode fiche :

HL

Code

Nom

Type

SAUVEGARDER

SUPPRIMER

CHERCHER

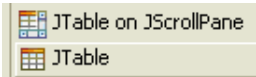
QUITTER

Les boutons ne correspondant pas au choix de l'utilisateur sont temporairement désactivés. Cette gestion des boutons permet de ne créer qu'une seule fenêtre pour les différentes actions.

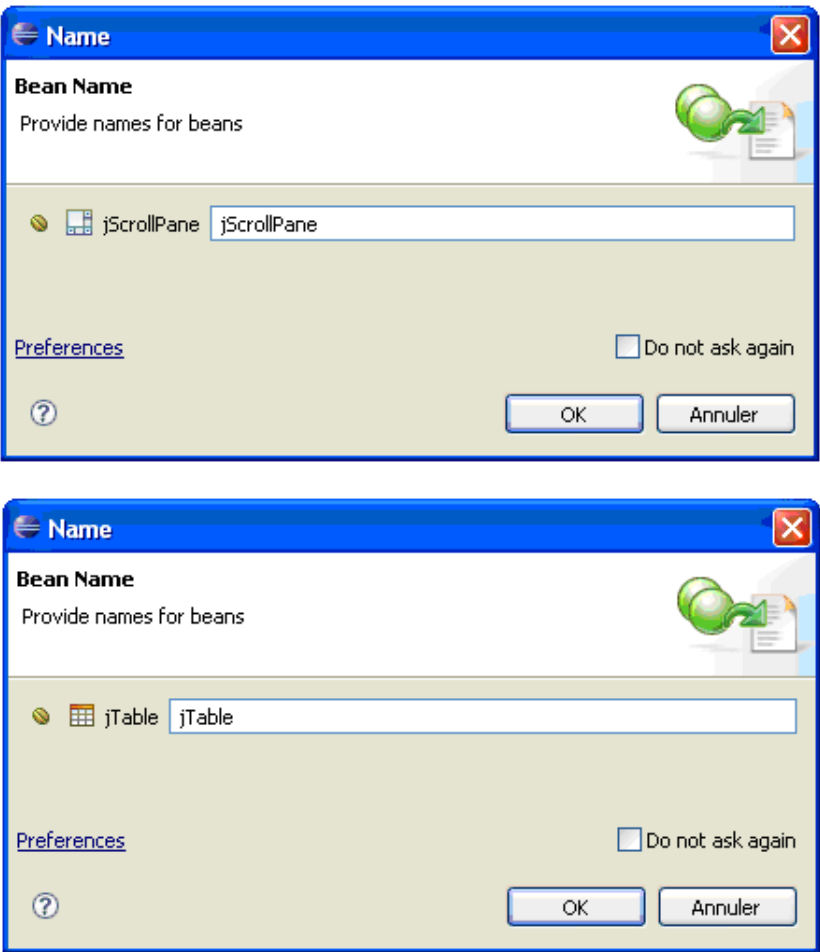
2. Maquette en mode table

- Ouvrez la classe **FenTableClient** à partir du paquetage **dialogue** avec Visual Editor.

Cette maquette nécessite l'utilisation d'un nouveau composant de type **JTable**. Nous avons le choix entre deux composants, le premier avec ascenseur, le second sans.



- Agrandissez la fenêtre de sorte qu'elle puisse accueillir un tableau de trois colonnes (voir la figure précédente).
- Sélectionnez dans la palette, le composant **JTable on JScrollPane** (avec ascenseur) et déposez-le dans la fenêtre.



Le composant **JTable** est également créé et contenu dans le composant **JScrollPane**.

- Poursuivez la construction de la maquette en renommant les composants comme indiqué ci-après.

Type de composant	Nom
JLabel	titre_Label
JTable	laTable_Table

JButton	btn_Ajouter, btn_Supprimer, btn_Modifier, btn_Rechercher, btn_Quitter
---------	--

Les méthodes pour les boutons ont été renommées :

getBtn_Ajouter(), getBtn_Supprimer(), getBtn_Modifier(), getBtn_Rechercher(), getBtn_Quitter()

Pour gagner du temps, laissez le nom et la méthode proposés par défaut par Eclipse pour le panneau : **jContentPane**, **getJContentPane()**. Idem pour **jScrollPane** et **getJScrollPane()**.

- Ajoutez les accesseurs et les mutateurs avec l'aide d'Eclipse (cf. chapitre Gestion de la connexion).

La conception d'une maquette a été déjà amplement détaillée. Nous ne nous attarderons plus sur le code graphique.

- Lancez la classe **FenTableClient** pour vérifier son aspect.

Les boutons sont pour l'instant inactifs et la table de données vide.

3. Maquette en mode fiche

Cette fenêtre permet d'afficher un enregistrement en mode liste.

- Avec Visual Editor, personnalisez la classe graphique **FenFicheClient** telle que présentée dans la section Fonctionnalités de ce chapitre.
- Renommez seulement les noms des boutons et leurs méthodes getXXX() :

- **btn_Ajouter, btn_Supprimer, btn_Rechercher, btn_Quitter**
- **getBtn_Ajouter(), getBtn_Supprimer(), getBtn_Rechercher(), getBtn_Quitter()**

Reportez-vous au code complet de cette classe à la fin de cette section pour les dimensions et le positionnement des composants.

- Ajoutez les accesseurs et les mutateurs avec l'aide d'Eclipse.
- Lancez la classe **FenFicheClient** pour vérifier son aspect.

Certains traitements en mode fiche nécessitent que des champs de saisie soient accessibles alors que d'autres doivent être verrouillés. Par exemple, on peut décider qu'une demande de suppression entraîne un verrouillage des champs **nom** et **type**, seul, le champ code restant accessible.

- Modifiez la portée des accesseurs aux champs de saisie en changeant le mot clé **private** par **public**.

```
public JTextField getJTxCCode() {
    ...
}
public JTextField getJTxTNom() {
    ...
}
return jTxTNom;
}
public JTextField getJTxTType() {
    ...
}
```

La fenêtre **FenFicheClient** n'est visible qu'à partir de la fenêtre **FenTableClient** et plus précisément lors de l'activation des boutons de celle-ci hormis bien sûr le bouton **QUITTER**.



La fenêtre **FenFicheClient** est alors ouverte, certains boutons étant désactivés en fonction du choix de l'utilisateur. Par exemple, pour un ajout :



L'idée est de créer une instance de **FenFicheClient** en même temps qu'une instance de la fenêtre **FenTableClient** est créée, la première étant alors non visible mais accessible par un handle.

- Modifiez la visibilité de la fenêtre **FenFicheClient** lors de sa création.

```
private void initialize() {
    ...
    this.setVisible(false);
}
```

- Ajoutez les méthodes gérant l'état des boutons.

```
public JButton setBtn_EnregistrerActif() {
    btn_Enregistrer.setEnabled(true);
    return btn_Enregistrer;
}
public JButton setBtn_EnregistrerNonActif() {
    btn_Enregistrer.setEnabled(false);
    return btn_Enregistrer;
}

public JButton setBtn_SupprimerActif() {
    btn_Supprimer.setEnabled(true);
    return btn_Supprimer;
}
public JButton setBtn_SupprimerNonActif() {
    btn_Supprimer.setEnabled(false);
    return btn_Supprimer;
}

public JButton setBtn_RechercherActif() {
    btn_Rechercher.setEnabled(true);
    return btn_Rechercher;
}
public JButton setBtn_RechercherNonActif() {
    btn_Rechercher.setEnabled(false);
    return btn_Rechercher;
}
```

4. Activation de la fenêtre client en mode table

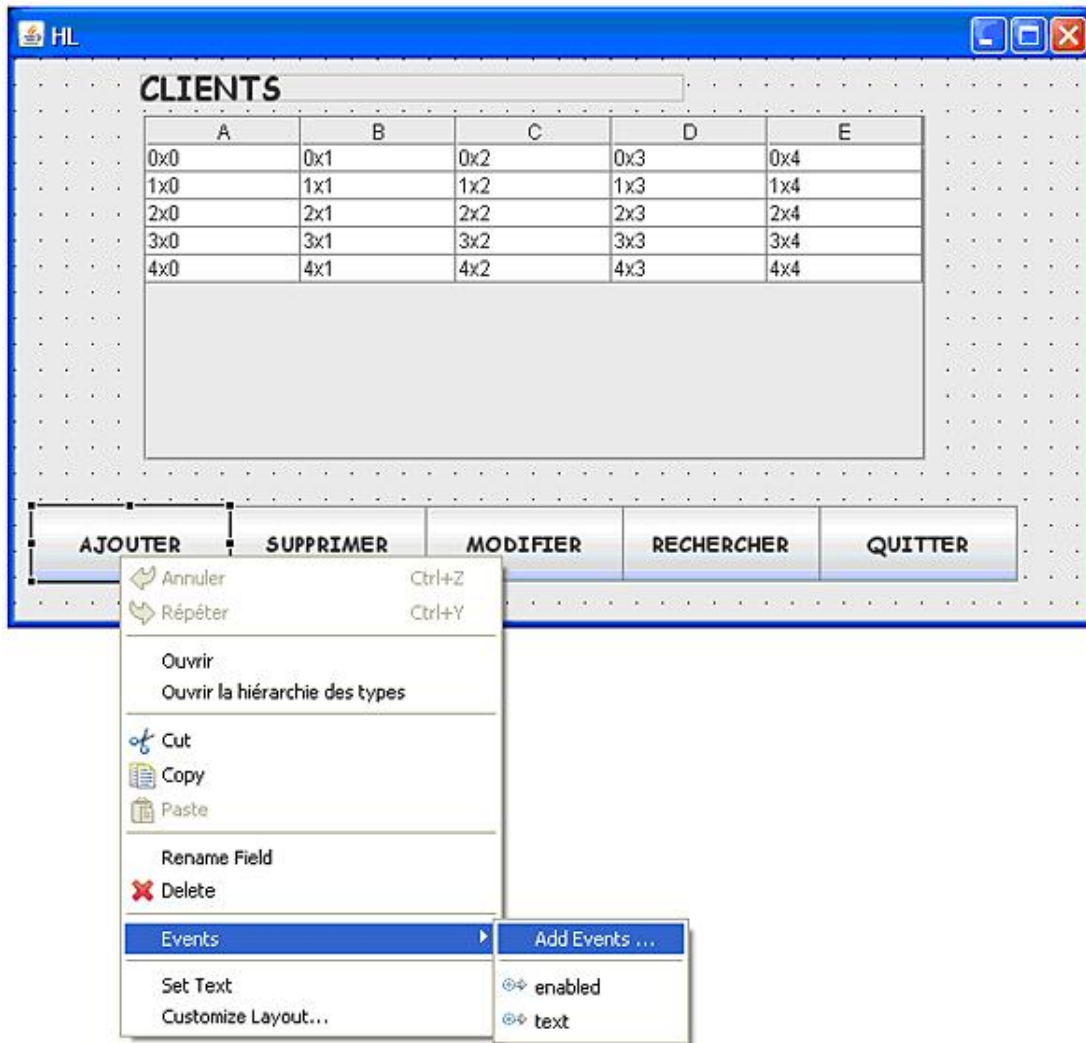
Nous allons rendre actifs tous les boutons de la fenêtre **FenTableClient**.

- Ouvrez la classe **FenTableClient** et insérez auparavant une nouvelle propriété.

```
private FenFicheClient laFiche_client = new FenFicheClient();
```

On peut procéder autrement, par exemple en instanciant la fenêtre **FenFicheClient** dans chaque bouton mais cela alourdit le code et multiplie les instantiations.

- Pour chaque bouton, ajoutez le code pour gérer l'événement clic de souris. Utilisez Eclipse pour vous faciliter la tâche à moins que vous ne préfériez, à ce stade de l'étude, écrire vous-même le code.



- Modifiez ensuite les méthodes agissant sur l'état des boutons : **getBtn_Ajouter()**, **getBtn_Supprimer()**, **getBtn_Modifier()**, **getBtn_Rechercher()**.

```
private JButton getBtn_Ajouter() {
    if (btn_Ajouter == null) {
        btn_Ajouter = new JButton();
        btn_Ajouter.setBounds(new Rectangle(10, 259, 115, 44));
        btn_Ajouter.setText("AJOUTER");
        btn_Ajouter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Ajouter.setName("");
        btn_Ajouter.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                // préparation de la fenêtre en mode fiche
                // 1. on ferme la fenêtre TableClient
                FenTableClient.this.dispose();
                laFiche_client.setBtn_EnregistrerActif();
                // 2. on désactive le bouton Supprimer de la fiche client
                laFiche_client.setBtn_SupprimerNonActif();
                // 3. on désactive le bouton Chercher de la fiche client
                laFiche_client.setBtn_RechercherNonActif();
                laFiche_client.setVisible(true);
            }
        });
    }
    return btn_Ajouter;
}
```



```

    }

    private JButton getBtn_Supprimer() {
        if (btn_Supprimer == null) {
            ...
            btn_Supprimer.addMouseListener(new java.awt.event.MouseAdapter() {
                public void mouseClicked(java.awt.event.MouseEvent e) {
                    FenTableClient.this.dispose();
                    laFiche_client.setBtn_EnregistrerNonActif();
                    laFiche_client.setBtn_RechercherNonActif();
                    laFiche_client.getJTxTCode().setEditable(true);
                    // les champs suivants sont masqués
                    laFiche_client.getJTxTNom().setVisible(false);
                    laFiche_client.getJTxTType().setVisible(false);
                    laFiche_client.setVisible(true);
                }
            });
        }
        return btn_Supprimer;
    }

    private JButton getBtn_Modifier() {
        if (btn_Modifier == null) {
            ...
            btn_Modifier.addMouseListener(new java.awt.event.MouseAdapter() {
                public void mouseClicked(java.awt.event.MouseEvent e) {
// rien pour l'instant, les modifications seront gérées différemment
                }
            });
            btn_Modifier.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        }
        return btn_Modifier;
    }

    private JButton getBtn_Rechercher() {
        if (btn_Rechercher == null) {
            ...
            btn_Rechercher.addMouseListener(new java.awt.event.MouseAdapter() {
                public void mouseClicked(java.awt.event.MouseEvent e) {
                    FenTableClient.this.dispose();
                    laFiche_client.setBtn_EnregistrerNonActif();
                    laFiche_client.setBtn_SupprimerNonActif();
                    laFiche_client.setVisible(true);
                }
            });
        }
        return btn_Rechercher;
    }
}

```

Code de la maquette de la fenêtre **FenTableClient** version 1.

```

package dialogue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.Rectangle;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.Font;

public class FenTableClient extends JFrame {
    // Propriétés
    private FenFicheClient laFiche_client = new FenFicheClient();
}

```

```

private static final long serialVersionUID = 1L;
private JPanel jContentPane = null;
private JScrollPane jScrollPane = null;
private JTable laTable_Table = null;
private JLabel titre_Label = null;
private JButton btn_Ajouter = null;
private JButton btn_Supprimer = null;
private JButton btn_Modifier = null;
private JButton btn_Rechercher = null;
private JButton btn_Quitter = null;

// Constructeur
public FenTableClient() {
    super();
    initialize();
}

// Méthodes
private void initialize() {
    this.setSize(631, 360);
    this.setContentPane(getJContentPane());
    this.setTitle("HL");
    this.setLocationRelativeTo(null);
    this.setVisible(true);
}

private JPanel getJContentPane() {
    if (jContentPane == null) {
        titre_Label = new JLabel();
        titre_Label.setBounds(new Rectangle(73, 9, 315, 16));
        titre_Label.setFont(new Font("Comic Sans MS", Font.BOLD, 18));
        titre_Label.setText("CLIENTS");
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
        jContentPane.add(jScrollPane, null);
        jContentPane.add(titre_Label, null);
        jContentPane.add(btn_Ajouter, null);
        jContentPane.add(btn_Supprimer, null);
        jContentPane.add(btn_Rechercher, null);
        jContentPane.add(btn_Quitter, null);
        jContentPane.add(btn_Modifier, null);
    }
    return jContentPane;
}

private JScrollPane getJScrollPane() {
    if (jScrollPane == null) {
        jScrollPane = new JScrollPane();
        jScrollPane.setBounds(new Rectangle(75, 33, 453, 200));
        jScrollPane.setViewportView(getJTable());
    }
    return jScrollPane;
}

private JTable getJTable() {
    if (laTable_Table == null) {
        laTable_Table = new JTable();
    }
    return laTable_Table;
}

private JButton getBtn_Ajouter() {
    if (btn_Ajouter == null) {
        btn_Ajouter = new JButton();
        btn_Ajouter.setBounds(new Rectangle(10, 259, 115, 44));
        btn_Ajouter.setText("AJOUTER");
        btn_Ajouter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Ajouter.setName("");
    }
}

```

```

        btn_Ajouter.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                // préparation de la fenêtre en mode fiche
                // 1. on ferme la fenêtre TableClient
                FenTableClient.this.dispose();
                laFiche_client.setBtn_EnregistrerActif();
                // 2. on désactive le bouton Supprimer de la fiche client
                laFiche_client.setBtn_SupprimerNonActif();
                // 3. on désactive le bouton Chercher de la fiche client
                laFiche_client.setBtn_RechercherNonActif();
                laFiche_client.setVisible(true);
            }
        });
    }
    return btn_Ajouter;
}

private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        btn_Supprimer = new JButton();
        btn_Supprimer.setBounds(new Rectangle(124, 259, 115, 44));
        btn_Supprimer.setText("SUPPRIMER");
        btn_Supprimer.setName("");
        btn_Supprimer.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Supprimer.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                FenTableClient.this.dispose();
                laFiche_client.setBtn_EnregistrerNonActif();
                laFiche_client.setBtn_RechercherNonActif();
                laFiche_client.getJTxTCode().setEditable(true);
                // on rend les champs suivants non éditables
                laFiche_client.getJTxTNom().setEditable(false);
                laFiche_client.getJTxTType().setEditable(false);
                laFiche_client.setVisible(true);
            }
        });
    }
    return btn_Supprimer;
}

private JButton getBtn_Modifier() {
    if (btn_Modifier == null) {
        btn_Modifier = new JButton();
        btn_Modifier.setBounds(new Rectangle(238, 259, 115, 44));
        btn_Modifier.setName("");
        btn_Modifier.setText("MODIFIER");
        btn_Modifier.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Modifier.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                // rien pour l'instant, les modifications seront gérées différemment
            }
        });
    }
    return btn_Modifier;
}

private JButton getBtn_Rechercher() {
    if (btn_Rechercher == null) {
        btn_Rechercher = new JButton();
        btn_Rechercher.setBounds(new Rectangle(352, 259, 115, 44));
        btn_Rechercher.setName("");
        btn_Rechercher.setText("RECHERCHER");
        btn_Rechercher.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Rechercher.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                FenTableClient.this.dispose();
                laFiche_client.setBtn_EnregistrerNonActif();
                laFiche_client.setBtn_SupprimerNonActif();
                laFiche_client.setVisible(true);
            }
        });
    }
}

```

```

    });
}
return btn_Rechercher;
}

private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(466, 259, 115, 44));
        btn_Quitter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
        btn_Quitter.setText("QUITTER");
        btn_Quitter.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                FenTableClient.this.dispose();
            }
        });
    }
}
return btn_Quitter;
}
}

```

Code complet de la maquette de la fenêtre **FenFicheClient** version 1.

```

package dialogue;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;
import java.awt.Rectangle;
import javax.swing.JLabel;
import javax.swing.JButton;

public class FenFicheClient extends JFrame {
    // Propriétés
    private static final long serialVersionUID = 1L;
    private JPanel jContentPane = null;
    private JTextField jTxTCode = null;
    private JTextField jTxTNom = null;
    private JTextField jTxTType = null;
    private JLabel jLabelCode = null;
    private JLabel jLabelNom = null;
    private JLabel jLabelType = null;
    private JButton btn_Enregistrer = null;
    private JButton btn_Quitter = null;
    private JButton btn_Supprimer = null;
    private JButton btn_Rechercher = null;

    // Constructeur
    public FenFicheClient() {
        super();
        initialize();
    }

    // Méthodes
    private void initialize() {
        this.setSize(579, 344);
        this.setContentPane(getJContentPane());
        this.setTitle("HL");
        this.setLocationRelativeTo(null);
        this.setVisible(false);
    }
    private JPanel getJContentPane() {
        if (jContentPane == null) {
            jLabelType = new JLabel();
            jLabelType.setBounds(new Rectangle(72, 99, 76, 16));
            jLabelType.setText("Type");
            jLabelNom = new JLabel();

```

```

        jLabelNom.setBounds(new Rectangle(72, 77, 78, 16));
        jLabelNom.setText("Nom");
        jLabelCode = new JLabel();
        jLabelCode.setBounds(new Rectangle(72, 53, 77, 16));
        jLabelCode.setText("Code");
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
        jContentPane.add(getJTxTCode(), null);
        jContentPane.add(getJTxTNom(), null);
        jContentPane.add(getJTxTType(), null);
        jContentPane.add(jLabelCode, null);
        jContentPane.add(jLabelNom, null);
        jContentPane.add(jLabelType, null);
        jContentPane.add(getBtn_Enregistrer(), null);
        jContentPane.add(getBtn_Quitter(), null);
        jContentPane.add(getBtn_Supprimer(), null);
        jContentPane.add(getBtn_Rechercher(), null);
    }
    return jContentPane;
}

public JTextField getJTxTCode() {
    if (jTxTCode == null) {
        jTxTCode = new JTextField();
        jTxTCode.setBounds(new Rectangle(151, 51, 125, 20));
    }
    return jTxTCode;
}

public JTextField getJTxTNom() {
    if (jTxTNom == null) {
        jTxTNom = new JTextField();
        jTxTNom.setBounds(new Rectangle(152, 75, 124, 20));
    }
    return jTxTNom;
}

public JTextField getJTxTType() {
    if (jTxTType == null) {
        jTxTType = new JTextField();
        jTxTType.setBounds(new Rectangle(153, 100, 124, 20));
    }
    return jTxTType;
}

private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        btn_Enregistrer = new JButton();
        btn_Enregistrer.setBounds(new Rectangle(19, 237, 126, 41));
        btn_Enregistrer.setText("SAUVEGARDER");
    }

    return btn_Enregistrer;
}

private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        btn_Supprimer = new JButton();
        btn_Supprimer.setBounds(new Rectangle(144, 237, 138, 41));
        btn_Supprimer.setText("SUPPRIMER");
    }
    return btn_Supprimer;
}

private JButton getBtn_Rechercher() {
    if (btn_Rechercher == null) {
        btn_Rechercher = new JButton();
        btn_Rechercher.setBounds(new Rectangle(281, 237, 126, 41));
        btn_Rechercher.setText("CHERCHER");
    }
}

```

```

        return btn_Rechercher;
    }

    // GESTION DE L'ACCESSIBILITE DES BOUTONS EN FONCTION DES CHOIX
    // -----
    // Ces méthodes rendent actifs/inactifs les boutons
    // lors d'un choix d'action effectué à partir de la fenêtre TableClient
    public JButton setBtn_EnregistrerActif() {
        btn_Enregistrer.setEnabled(true);
        return btn_Enregistrer;
    }
    public JButton setBtn_EnregistrerNonActif() {
        btn_Enregistrer.setEnabled(false);
        return btn_Enregistrer;
    }

    public JButton setBtn_SupprimerActif() {
        btn_Supprimer.setEnabled(true);
        return btn_Supprimer;
    }
    public JButton setBtn_SupprimerNonActif() {
        btn_Supprimer.setEnabled(false);
        return btn_Supprimer;
    }

    public JButton setBtn_RechercherActif() {
        btn_Rechercher.setEnabled(true);
        return btn_Rechercher;
    }
    public JButton setBtn_RechercherNonActif() {
        btn_Rechercher.setEnabled(false);
        return btn_Rechercher;
    }

    private JButton getBtn_Quitter() {
        if (btn_Quitter == null) {
            btn_Quitter = new JButton();
            btn_Quitter.setBounds(new Rectangle(406, 237, 126, 41));
            btn_Quitter.setText("QUITTER");
        }
        return btn_Quitter;
    }
}

```

Gestion de l'affichage des données

Dans cette section, nous abordons les traitements. Ceux-ci constituent la partie la plus délicate du développement mais aussi la plus intéressante. Dans le cadre de notre étude nous nous limitons à la gestion des clients pour les opérations suivantes :

- affichage ;
- modification ;
- ajout ;
- suppression ;
- recherche.

Pour toutes ces opérations, la base de données MySQL est sollicitée soit pour restituer les enregistrements soit pour sauvegarder toute modification ou ajout réalisé par le biais des objets de l'application. Une connaissance basique du langage SQL s'avère nécessaire pour construire les requêtes. Cette section permet aussi de rendre opérationnelles les fenêtres clients.

1. Récupération des enregistrements

a. Phase préparatoire

À la première utilisation de l'application, deux cas de figure peuvent se présenter pour la base de données : soit elle est vide, soit elle contient déjà des enregistrements. Il conviendrait d'en tenir compte et d'afficher le mode adéquat, soit la fiche pour un ajout ou la table pour visualiser les enregistrements lors de la première activation du bouton **CLIENTS** de la fenêtre menu, selon l'état initial de la base de données. Pour aller à l'essentiel, nous allons considérer le deuxième cas et réutiliser les enregistrements de la base créée au chapitre Base de données MySQL. Là aussi, nous avons privilégié la compréhension en créant une base de données minimaliste de quatre tables, la table client ne possédant dans le même ordre d'idée que trois champs.

Revoyons les enregistrements déjà existants.

- Lancez le serveur Wamp.
- Dans la barre des tâches, cliquez sur l'icône de Wamp.
- Cliquez sur **phpMySQL**.
- Choisissez la base de données **bdjavacagou** puis cliquez sur la table **client**.



La structure de la table **client** est alors présentée.

	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra	Action					
<input type="checkbox"/>	Code	varchar(5)	latin1_swedish_ci		Non								
<input type="checkbox"/>	Nom	varchar(20)	latin1_swedish_ci		Non								
<input type="checkbox"/>	Type	varchar(15)	latin1_swedish_ci		Non								

- Cliquez sur l'onglet **Afficher**. Les enregistrements apparaissent.

←T→	Code	Nom	Type
<input type="checkbox"/>	BO1	BAUMER SA	Société
<input type="checkbox"/>	DR1	DROUX	Particulier
<input type="checkbox"/>	LA1	LANGLE	Artisan

Nous aimerions qu'à l'ouverture de la fenêtre **FenTableClient**, les données de la base soient présentées en mode table.

Pour y parvenir, il faut passer par plusieurs étapes :

- Interrogation de la base avec une requête SQL.
- Transfert du jeu d'enregistrements obtenu dans un objet Java de type collection.
- Affichage des données de cet objet dans un composant contenu dans une IHM.

Considérons la première étape. Quelle classe comporte la méthode d'interrogation ? En référence à l'analyse réalisée, il s'agit de la classe **Client**. Celle-ci étend la classe **Personne**. Nous allons donc commencer par apporter les modifications concernant ces classes.

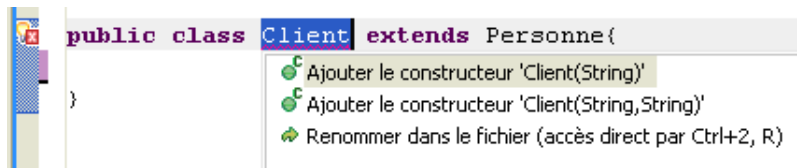
- Revenez sous Eclipse et ouvrez la classe **Personne** avec l'éditeur Java. Complétez son code.

```
public abstract class Personne {
    private String code;
    private String nom;
    // 1er constructeur
    public Personne (String vCode, String vNom){
        code = vCode;
        nom = vNom;
    }
    // 2ème constructeur
    public Personne (String vCode){
        code = vCode;
    }
    public String getCode(){
        return code;
    }
    public String getNom(){
        return nom;
    }
    public void setCode(String c){
        code = c;
    }
    public void setNom(String n){
        nom = n;
    }
}
```




Deux constructeurs sont désormais présents. Nous verrons leur utilité plus loin. Rappel : le mot clé **abstract** signifie que cette classe ne peut être instanciée.

Eclipse signale une erreur dans la classe **Client**. Ceci est dû au fait que le constructeur de la classe mère vient d'être redéfini. La définition des constructeurs des classes dérivées devient alors obligatoire.



Nous allons maintenant nous occuper de la classe **Client**. Tel que précisé au chapitre Analyse, c'est elle qui possède toutes les méthodes agissant sur la base de données. Cette classe spécialise la classe **Personne**.

- Ouvrez la classe **Client** avec l'éditeur Java et complétez son code. Il s'agit pour l'instant de la structure générale.

```
package entite;

public class Client extends Personne{
    private String type;
    // 1er Constructeur
    public Client(String vCode, String vNom, String vType){
        super(vCode, vNom);
        type = vType;
    }
    // 2ème Constructeur
    public Client(String vCode){
        super(vCode);
    }
    public String getType(){
        return type;
    }
    public void setType(String vTitre){
        type = vTitre;
    }
    // Ajout d'un nouveau client dans la BD
    // -----
    public void creerCRUD_Client(Client leClient){
    }
    // Modification d'un client dans la BD
    // -----
    public void modifierCRUD_Client(Client leClient){
    }
    // Suppression d'un client dans la BD
    // -----
    public void supprimerCRUD_Client(Client leClient){
    }
    // Chercher un ou plusieurs clients dans la BD
    // -----
    public int chercherCRUD_Clients(Client lesClients){
        int nbClients = 0;
        return nbClients;
    }
}
```



Cette classe possède aussi deux constructeurs. Le premier pour créer une personne avec toutes ses propriétés, le second qui est utilisé pour la recherche. Dans un souci de simplicité, la recherche est limitée au code. Outre les accesseurs et les mutateurs, les méthodes agissant sur la base de données sont déclarées. Il reste à les définir...

b. Interrogation de la base avec une requête SQL

Pour agir sur une base de données avec JDBC, deux opérations principales doivent être réalisées :

- connexion au serveur et à la base de données ;
- exécution des requêtes SQL.

Dans notre projet, la première opération est déjà prise en charge par la classe **ControleConnexion**. Celle-ci importe les classes **Connection Driver** et **Manager** pour effectuer les opérations suivantes :

- enregistrement et chargement du driver auprès du gestionnaire de pilotes ;
- connexion et identification au serveur et à la base de données.

À l'issue de cette étape, si tout s'est bien passé, un objet **Connection** unique est créé ou plus précisément une instance de la classe **DriverManager** qui implémente l'interface **Connection** (cf. le chapitre Base de données MySQL).

La deuxième opération concerne la classe **Client** et se décompose en plusieurs traitements :

- création d'un état ou statement pour les requêtes SQL ;
- exécution des requêtes SQL ;
- parcours des jeux d'enregistrements au besoin s'il s'agit de sélections ;
- fermeture des jeux d'enregistrements ;
- fermeture du statement ;
- déconnexion du serveur en fermant la connexion.

Plusieurs importations doivent être ajoutées à la classe **Client** pour obtenir la connexion à la base de données, gérer les requêtes SQL et leur contenu.

■ Ajoutez les importations.

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;

import controle.*;
```

Pour établir le lien avec la classe **ControleGestion**, une propriété de ce type est déclarée et initialisée. Par le biais de l'instance obtenue, la connexion statique est récupérée et permet l'initialisation d'une deuxième propriété de type **Connection**.

■ Complétez les propriétés pour récupérer la connexion statique.

```
private ControleConnexion leControleConnexion = new ControleConnexion();
private Connection laConnexion = leControleConnexion.getConnexion();
```

Nous pouvons maintenant écrire la méthode qui sélectionne les enregistrements à partir d'une requête SQL. Nous l'écrivons de sorte qu'elle puisse servir aussi pour des recherches paramétrées.

```
public int chercherCRUD_Clients(Client lesClients){
    String leCode = lesClients.getCode();
    String requete = null;
    try {
        // on facilite la saisie avec le joker SQL "%"
        leCode = leCode + "%";
        requete = "SELECT * FROM client WHERE Code LIKE '"+leCode+"'"
    }
```

```

        + " ORDER BY code, nom";

        Statement state = laConnexion.createStatement();
        ResultSet jeuEnregistrements = state.executeQuery(requete);
    }

    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Problème lors de la recherche.", "",
            JOptionPane.ERROR_MESSAGE);
    }
}

```



Le joker % remplace n'importe quelle suite de caractères. La saisie est ainsi simplifiée pour les recherches.

Attention à ne pas employer le signe "=" qui n'admet qu'un enregistrement en retour.

Il faut être vigilant quant à la syntaxe pour l'écriture des requêtes, par exemple l'oubli d'un espace devant l'instruction **LIKE** provoque une erreur. Dans la table **client** de la base MySQL, le champ **Code** est déclaré clé primaire. L'utilisateur est ainsi averti s'il saisi un code déjà existant. Respectez scrupuleusement l'orthographe et la casse des noms des tables et des champs de la base de données MySQL.

Un objet **Statement** est créé. Il fournit les méthodes pour exécuter les requêtes SQL telles que **executeQuery()** pour les sélections, **executeUpdate()** pour les modifications et **execute()** pour tout autre ordre SQL. Il existe d'autres variantes de la classe **Statement** : la classe **PreparedStatement** pour des requêtes paramétrées et la classe **CallableStatement** pour les procédures stockées. Pour plus d'informations sur leur emploi, consulter l'aide de SUN.

- Lancez l'application pour tester les modifications apportées au code.

c. Transfert des enregistrements dans un objet Collection

Dans la fenêtre **FenTableClient**, aucun enregistrement n'est visible. Cela est normal. Il faut passer à la deuxième étape telle que mentionnée plus haut dans la phase préparatoire : il s'agit maintenant de transférer les enregistrements dans une structure d'accueil appropriée.

Les enregistrements peuvent être gérés de diverses manières avec Java, soit par le biais d'un tableau statique (la taille est fixe et définie à la création) à deux dimensions soit à l'aide de vecteurs (**Vector**) ou de liste de tableaux (**ArrayList**) qui sont tous deux des collections, sorte de "supers" tableaux dynamiques (la taille n'est pas figée et peut varier en fonction des besoins). Ils sont par ailleurs très proches en terme de fonctionnalité et possèdent en outre davantage de méthodes que les tableaux, ce qui facilite la manipulation des données. Pour notre projet, nous utilisons les vecteurs.

Positionnement de la classe **Vector**.

```

java.lang.Object
├── java.util.AbstractCollection
│   ├── java.util.AbstractList
│       └── java.util.Vector

```



La classe **Vector** est plus ancienne que la classe **ArrayList**. Celle-ci est plus rapide mais **Vector** est **thread safe**, ce qui signifie qu'il est possible d'accéder aux données d'un vecteur à partir de plusieurs processus.

Une fois le transfert des enregistrements effectué dans un vecteur, la manipulation des données se fait par le biais de celui-ci. Pour qu'il soit accessible par la classe **FenTableClient**, le plus simple est d'ajouter à la classe **Client** de nouvelles propriétés de type Vector.

- Insérez les propriétés suivantes dans la classe **Client** sans oublier l'importation attendue :

```

...
import java.util.Vector;
...
private Vector<Vector> tabLignes = new Vector<Vector>();
private Vector<String> nomColonnes = new Vector<String>();

```



La classe **Vector** permet de stocker des objets mais il convient de préciser leur type : **tabLignes** est un vecteur

de vecteurs, **nomColonnes** est un vecteur contenant des chaînes de caractères.

- Complétez ensuite la méthode **chercherCRUD_Clients()** de la classe **Client**.

```
public int chercherCRUD_Clients(Client lesClients){
    String leCode = lesClients.getCode();
    int i, nbClients=0;
    String requete = null;
    try {
        leCode = leCode + "%";
        requete = "SELECT * FROM client WHERE Code LIKE '"+leCode+"'"
        + " ORDER BY code, nom";
        Statement state = laConnexion.createStatement();
        ResultSet jeuEnregistrements = state.executeQuery(requete);
        ResultSetMetaData infojeuEnregistrements = jeuEnregistrements.getMetaData();

        for( i=1; i <= infojeuEnregistrements.getColumnCount(); i++)
        { nomColonnes.add(infojeuEnregistrements.getColumnLabel(i));}

        while(jeuEnregistrements.next()) {
            Vector<String> ligne = new Vector<String>();
            for(i=1; i <= infojeuEnregistrements.getColumnCount(); i++) {
                String chaine_champ = jeuEnregistrements.getString(i);
                ligne.add(chaine_champ);
            }
            tabLignes.add(ligne);
            nbClients = nbClients + 1;
        }

        jeuEnregistrements.close();
        state.close();
    }

    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Problème lors de la recherche.", "",
            JOptionPane.ERROR_MESSAGE);
    }
    return nbClients;
}
```

L'interface **ResultSetMetaData** comporte de nombreuses méthodes permettant d'obtenir des informations sur le jeu d'enregistrements issu d'un SELECT : nombre de colonnes, libellé d'une colonne, table d'origine, etc. Une instance de **ResultSetMetaData** est obtenue à partir d'un objet instance de la classe **ResultSet** avec la méthode **getMetaData()**.

Le nombre de colonnes est récupéré avec la méthode **getColumnCount()** de l'objet **infojeuEnregistrements** de type **ResultSetMetaData** et sert de condition d'arrêt à la première boucle **for**. Celle-ci est simple. Les titres des colonnes sont récupérés l'un après l'autre avec la méthode **getColumnLabel()** et rangés dans le vecteur **nomColonnes**. Avec le jeu d'enregistrements présent dans la base, on obtient le résultat suivant :

nomColonnes

Code	Nom	Type
------	-----	------

La structure **while** est plus complexe. Cette fois-ci, il faut parcourir le jeu d'enregistrements. Durant le parcours, pour chaque enregistrement, un vecteur nommé **ligne** est créé et reçoit les données de l'enregistrement courant l'une après l'autre. C'est le rôle du **for** imbriqué qui parcourt les colonnes pour chaque enregistrement.

Dès qu'un vecteur est rempli (à la sortie du **for**), il est ajouté au vecteur **tabLignes** qui est un vecteur contenant des vecteurs. À la sortie du **while**, on obtient le résultat suivant :

tabLignes				
	←	BO1	BAUMIER SA	Société
	←	DR1	DROUX	Particulier
	←	LA1	LANGLE	Artisan

Ne pas oublier de fermer le jeu d'enregistrements et le statement.

Pour finir, la méthode **chercherCRUD_Clients()** retourne le nombre d'enregistrements trouvés.

- Lancez l'application pour voir le résultat.


L'affichage en mode table reste désespérément vide car un vecteur ne peut être affiché directement dans une interface graphique.

2. Affichage des données dans un composant JTable

Pour afficher les données en mode table dans la fenêtre **FenTableClient**, il faut créer une nouvelle méthode qui retourne un composant **JTable** constitué des vecteurs provenant de la classe **Client**.

- Ouvrez la classe **FenTableClient** et ajoutez la méthode **remplirTable()**.

```
private JTable remplirTable(){
    String vCode = "";
    // tapez les premières lettres, par exemple "en"
    // puis appuyez simultanément sur les touches Ctrl et Espace
    entite.Client lesClients = new entite.Client(vCode);
    lesClients.chercherCRUD_Clients(lesClients);
    return new JTable(lesClients.getTabLignes(), lesClients.getNomColonnes());
}
```

 Le deuxième constructeur de la classe **Client** est utilisé. La recherche s'effectue sur le code ce qui correspond ici à la sélection de tous les enregistrements de la table **client** (utilisation du joker %). Par le biais de l'instance, les vecteurs contenant les données sont transmis au composant **JTable**.

Il est possible pour faciliter la compréhension d'écrire la dernière ligne différemment :

```
Vector<Vector> Lignes = lesClients.getTabLignes();
Vector<String> NomColonnes = lesClients.getNomColonnes();
return new JTable(Lignes, NomColonnes);
```

N'oubliez pas dans ce cas, de procéder à l'importation de la classe **Vector**. En cas d'oubli, Eclipse vous signalera une erreur et vous proposera l'importation de cette classe.

- Lancez l'application pour voir le résultat.

Toujours aucune donnée visible. Que manque-t-il ? Il reste à modifier l'initialisation du composant **JTable** dont le code est rappelé ci-après :

```
private JTable getJTable() {
    if (laTable_Table == null) {
        laTable_Table = new JTable();
    }
    return laTable_Table;
}
```

 La troisième ligne peut être supprimée puisqu'elle est prise en charge par la méthode **remplirTable()**.

- Effectuez les modifications.

```
private JTable getJTable() {if (laTable_Table == null) {
    laTable_Table = remplirTable();
}
return laTable_Table;
}
```

- Lancez à nouveau l'application.

Les données sont enfin visibles ainsi que les titres de colonnes.

Code	Nom	Type
BO1	BAUMIER SA	Société
DR1	DROUX	Particulier
LA1	LANGLE	Artisan

Classe **Client** version 2

```
package entite;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.Vector;

import controle.*;

import javax.swing.JOptionPane;

public class Client extends Personne{
    // Propriétés
    private String type;
    private ControleConnexion leControleConnexion = new ControleConnexion();
    private Connection laConnexion = leControleConnexion.getConnexion();

    private Vector<Vector> tabLignes = new Vector<Vector>();
    private Vector<String> nomColonnes = new Vector<String>();
    public Vector<Vector> getTabLignes(){
        return tabLignes;
    }
    public Vector<String> getNomColonnes(){
        return nomColonnes;
    }

    // 1er Constructeur
    public Client(String vCode, String vNom, String vType){
        super(vCode, vNom);
        type = vType;
    }
    // 2ème Constructeur
    public Client(String vCode){
        super(vCode);
    }

    public String getType(){return type;
    }
    public void setType(String vTitre){
        type = vTitre;
    }

    // Ajout d'un nouveau client dans la BD
    // -----
    public void creerCRUD_Client(Client leClient){
    }
    // Modification d'un client dans la BD
    // -----
    public void modifierCRUD_Client(Client leClient){
    }
    // Suppression d'un client dans la BD
    // -----
    public void supprimerCRUD_Client(Client leClient){
    }
    // Recherche d'un ou plusieurs clients dans la BD
    // -----
```

```

public int chercherCRUD_Clients(Client lesClients){
    String leCode = lesClients.getCode();
    int i, nbClients=0;
    String requete = null;

    try {
        leCode = leCode + "%";
        requete = "SELECT * FROM client WHERE Code LIKE '"+leCode+"'"
            + " ORDER BY code, nom";

        Statement state = laConnexion.createStatement();
        ResultSet jeuEnregistrements = state.executeQuery(requete);
        ResultSetMetaData infojeuEnregistrements = jeuEnregistrements.getMetaData();

        for( i=1; i <= infojeuEnregistrements.getColumnCount(); i++)
            nomColonnes.add(infojeuEnregistrements.getColumnLabel(i));
        while(jeuEnregistrements.next()) {
            Vector<String> ligne = new Vector<String>();
            for(i=1; i <= infojeuEnregistrements.getColumnCount(); i++) {
                String chaine_champ = jeuEnregistrements.getString(i);
                ligne.add(chaine_champ);
            }
            tabLignes.add(ligne);
            nbClients = nbClients + 1;
        }
        jeuEnregistrements.close();
        state.close();
    }
    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Problème lors de la recherche.", "",
            JOptionPane.ERROR_MESSAGE);
    }
    return nbClients;
}
}

```

Classe **Personne** version 2

```

package entite;

public abstract class Personne {
    private String code;
    private String nom;
    // 1er constructeur
    public Personne (String vCode, String vNom){
        code = vCode;
        nom = vNom;
    }
    // 2ème constructeur
    // pour les recherches
    public Personne (String vCode){
        code = vCode;
    }
    public String getCode(){
        return code;
    }
    public String getNom(){
        return nom;
    }
    public void setCode(String c){
        code = c;
    }
    public void setNom(String n){
        nom = n;
    }
}

```

```
package dialogue;

...

// Initialisation du composant JTable
private JTable getJTable() {
    if (laTable_Table == null) {
        laTable_Table = remplirTable();
    }
    return laTable_Table;
}

private JTable remplirTable(){
    String vCode = "";
    entite.Client lesClients = new entite.Client(vCode);
    lesClients.chercherCRUD_Clients(lesClients);
    return new JTable(lesClients.getTabLignes(),
        lesClients.getNomColonnes());
}

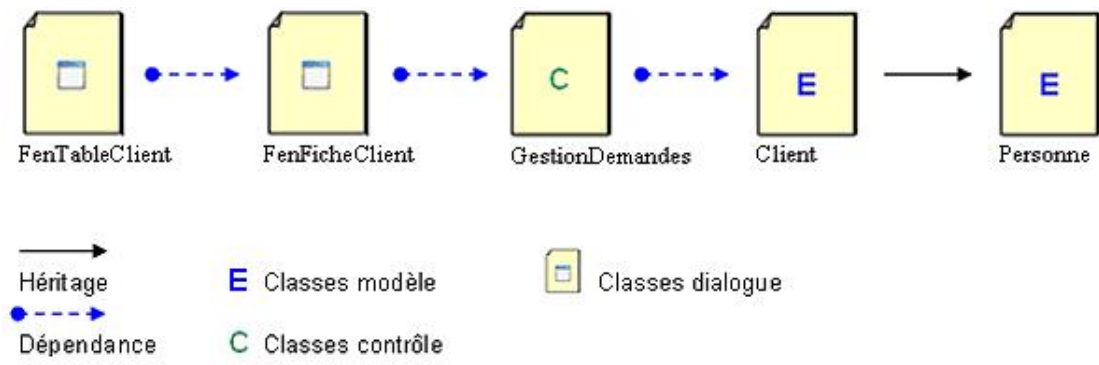
...
}
```


Gestion des traitements

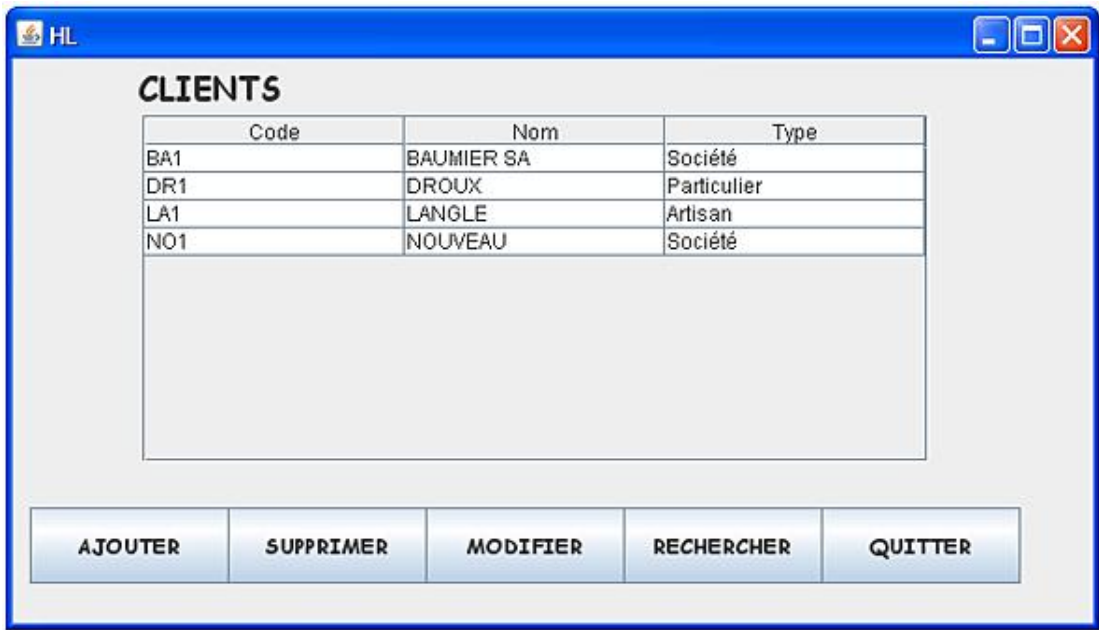
1. Ajout d'un enregistrement

Nous aimerions pouvoir tester assez rapidement l'ajout d'un client. Mais de la fenêtre présentant les données en mode table à la fenêtre d'ajout, plusieurs classes interviennent et nécessitent une modification de leur code.

Pour y voir un peu plus clair et en se référant au diagramme des classes (cf. chapitre Analyse), voici une vue simplifiée et personnelle des classes concernées par ce traitement.




La fenêtre issue de la classe **FenTableclient** présente les données en mode table et comporte les boutons permettant à l'utilisateur d'effectuer les traitements.



La fenêtre issue de la classe **FenFicheclient** présente les données en mode fiche. Elle est ouverte à partir des boutons de la fenêtre précédente. Quel que soit le traitement demandé, c'est la même fenêtre qui est présentée, certains boutons étant alors désactivés (cf. la section Affichage des données dans une table dans ce chapitre).


La classe **GestionDemandes** est de type contrôle. Elle existe déjà mais n'a pas encore été traitée. Elle assure la jonction entre l'IHM et les classes de type entité. Plus précisément, elle prend en compte toutes les demandes de l'utilisateur qui modifient l'état de la base de données et les transmet pour traitement à la classe **Client**.

La classe **Client** possède les méthodes agissant sur la base de données de type CRUD. Elle hérite en outre de la classe **Personne**.

 Cette répartition des tâches en couches présente un intérêt réel. Si le code des méthodes de la classe **Client** devait changer, les modifications à apporter aux autres classes seraient minimales, voire nulles.

Voici la démarche retenue pour gérer l'ajout d'un client :

1. La méthode **creerCRUD_Client()** de la classe **Client** est complétée. Elle assure la liaison avec la base de données et permet la sauvegarde.
2. Une méthode de la classe **GestionDemandes** est créée et nommée **demandeEnregistrerClient()**. Cette méthode contient un appel à la méthode **creerCRUD_Client()**.
3. La classe **FenFicheClient** est modifiée pour informer la classe **GestionDemandes** de la demande d'ajout.

 Les noms de certaines méthodes sont volontairement longs afin de faciliter la compréhension lors de la lecture du code.

- Ouvrez la classe **Client** et complétez le code de la méthode **creerCRUD_Client()**.

```
public void creerCRUD_Client(Client leClient){
    String requete = null;
    try {
        requete = "INSERT INTO client(Code, Nom, Type)" +
            " VALUES('"+super.getCode()+"', '"+super.getNom()+"', '"+type+"')";

        Statement state = laConnexion.createStatement();
        state.executeUpdate(requete);
        state.close();
    }
    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Ajout non effectué."
            + " Ce code client existe déjà.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        System.out.println("Insertion non effectuée");
    }
}
```

```
}  
}
```



La requête SQL est écrite de sorte à bien mettre en évidence le mécanisme d'héritage. Il est possible d'utiliser des variables pour une plus grande lisibilité. Exemple :

```
String leCode = super.getCode();  
String leNom = super.getNom();  
String leType = type;  
requete = "INSERT INTO client(Code, Nom, Type)" +  
        " VALUES('"+leCode+"', '"+leNom+"', '"+leType+"')";
```

- Ouvrez maintenant la classe **GestionDemandes** et créez la méthode **demandeEnregistrerClient()**.

```
public void demandeEnregistrerClient(String vCode, String vNom, String vType){  
    Client leClient = new Client(vCode, vNom, vType);  
    leClient.creerCRUD_Client(leClient);  
}
```



Cette méthode attend trois paramètres de type chaîne qui sont fournis lors de son appel à partir de la fenêtre **FenFicheClient**.

- Pour finir, ouvrez la classe **FenFicheClient**. Procédez auparavant à l'importation de la classe **GestionDemandes** et à la déclaration d'une nouvelle propriété pour établir le lien.

```
...  
import controle.GestionDemandes;  
...  
private GestionDemandes leClientGestionClientBD = new GestionDemandes();  
...
```

- Modifiez le code de la méthode **getBtn_Enregistrer()** pour transmettre la demande d'ajout à la classe **GestionDemandes**.

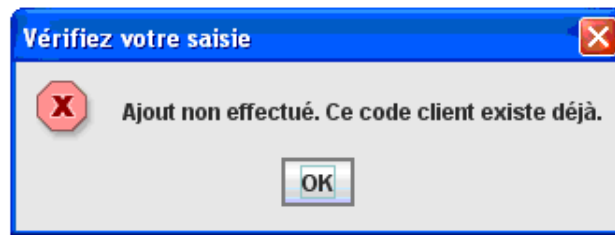
```
private JButton getBtn_Enregistrer() {  
    ...  
    if (btn_Enregistrer == null) {  
        btn_Enregistrer.addMouseListener(new java.awt.event.MouseAdapter() {  
            public void mouseClicked(java.awt.event.MouseEvent e) {  
                String vCode = jTxTCode.getText();  
                String vNom = jTxTNom.getText();  
                String vType = jTxTType.getText();  
                leClientGestionClientBD.demandeEnregistrerClient(vCode, vNom, vType);  
            }  
        });  
    }  
}
```



Les paramètres de l'utilisateur sont transmis à la méthode **demandeEnregistrerClient()** de la classe **GestionDemandes**.

- Lancez l'application et testez l'ajout d'un client.
- Effectuez un autre ajout en ressaisissant le même code pour vérifier l'interception de l'exception traitée dans la méthode **creerCRUD_Client()** de la classe **Client**.


Vous devriez obtenir le message suivant.



Ceci nous amène à mettre en place un autre contrôle élémentaire qui vérifie que le code de saisie a bien été saisi.

- Ajoutez ce contrôle sans oublier l'importation pour les boîtes de dialogue.

```
...
import javax.swing.JOptionPane;
...
private JButton getBtn_Enregistrer() {
    ...
    if (btn_Enregistrer == null) {
        btn_Enregistrer.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                String vCode = jTxTCode.getText();
                String vNom = jTxTNom.getText();
                String vType = jTxTType.getText();
                if (!vCode.equals("")) {
                    leClientGestionClientBD.demandeEnregistrerClient(vCode, vNom, vType);
                    jTxTCode.setText("");
                    jTxTNom.setText("");
                    jTxTType.setText("");
                    jTxTCode.requestFocusInWindow();
                }
                else
                {
                    JOptionPane.showMessageDialog(null, "La saisie du code client"
                        + " est obligatoire",
                        "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
                }
            }
        });
    }
}
```

 Si l'ajout s'est bien passé, les champs de saisie sont vidés et le focus est donné au champ **jTxTCode**. En cas de problème, des messages apparaissent. Les boîtes de dialogue sont ici réparties entre les deux classes **FenFicheClient** et **Client**.

Pour faciliter les nombreux tests qui vont suivre, nous allons de suite compléter le code du bouton **QUITTER** de la fenêtre **FenFicheClient**.

- Modifiez le code de la méthode.

```
private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {
        btn_Quitter = new JButton();
        btn_Quitter.setBounds(new Rectangle(406, 237, 126, 41));
        btn_Quitter.setText("QUITTER");
        btn_Quitter.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent e) {
                FenFicheClient.this.dispose();
                FenTableClient laTable_Client = new FenTableClient();
                laTable_Client.setVisible(true);
            }
        });
    }
    return btn_Quitter;
}
```

```
}
```



La nouvelle instance de **FenTableClient** présente le composant **JTable** avec les données réactualisées.

2. Suppression d'un enregistrement

La démarche retenue suit les mêmes étapes que pour l'ajout :

1. La méthode **supprimerCRUD_Client()** de la classe **Client** est complétée.
 2. Une méthode de la classe **GestionDemandes** est créée et nommée **demandeSupprimerClient()**. Cette méthode contient un appel à la méthode **supprimerCRUD_Client()**.
 3. La classe **FenFicheClient** est modifiée pour informer la classe **GestionDemandes** de la demande de suppression.
- Ouvrez la classe **Client** et complétez le code de la méthode **supprimerCRUD_Client()**.

```
public void supprimerCRUD_Client(Client leClient){
    String leCode = leClient.getCode();
    String requete = null;
    try {
        requete = "DELETE FROM client" +
            " WHERE Code = '"+leCode+"'";
        Statement state = laConnexion.createStatement();
        int nbEnregSup = state.executeUpdate(requete);
        if (nbEnregSup == 0){
            JOptionPane.showMessageDialog(null, "Aucune suppression effectuée."
                + " Ce code client n'existe pas.",
                "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        }
        else {
            state.close();
            JOptionPane.showMessageDialog(null, "Suppression du client code ["
                + leCode + "] effectuée");
        }
    }
    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Problème lors de la suppression.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
    }
}
```

- Ouvrez maintenant la classe **GestionDemandes** et créez la méthode **demandeSupprimerClient()**.

```
public void demandeSupprimerClient(String vCode){
    Client leClient = new Client(vCode);
    leClient.supprimerCRUD_Client(leClient);
}
```



Cette méthode attend un seul paramètre de type chaîne qui est fourni lors de son appel à partir de la fenêtre **FenFicheClient**.


- Pour finir, ouvrez la classe **FenFicheClient** et modifiez le code de la méthode **getBtn_Supprimer()** pour transmettre la demande de suppression à la classe **GestionDemandes**.

```
private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        ...
        public void actionPerformed(java.awt.event.ActionEvent e) {
```

```

String vCode = jTxTCode.getText();
if(!vCode.equals("")){
    int choix = JOptionPane.showConfirmDialog(null,
        "Voulez-vous supprimer la fiche du client code : " + vCode,"Suppression",
        JOptionPane.YES_NO_OPTION);
    if (choix==0){
        // demande de suppression
        leClientGestionClientBD.demandeSupprimerClient(vCode);
        jTxTCode.setText("");
        jTxTCode.requestFocusInWindow();
    }
}
else
{
    JOptionPane.showMessageDialog(null, "La saisie du code client"
        + " est obligatoire", "Vérifiez votre saisie",
        JOptionPane.ERROR_MESSAGE);
}
}
});
}
return btn_Supprimer;
}

```

 La suppression est toujours précédée d'une demande de confirmation. Après la suppression, le champ code est remis à vide, le focus est donné au champ **jTxTCode** et un message informe de la réussite de l'opération. D'autres messages sont adressés en cas de problème. Les boîtes de dialogue sont aussi réparties entre les deux classes **FenFicheClient** et **Client**.

Pour améliorer l'ergonomie, nous ajoutons la possibilité de supprimer une ligne sélectionnée dans le composant JTable. Deux avantages, les données sont visibles et gain de temps puisque le code n'est plus à saisir.

- Ouvrez la classe **FenTableClient** et complétez le code de la méthode **getBtn_Supprimer ()**.

```


private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        ...
        btn_Supprimer.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                // par défaut aucune ligne encore sélectionnée
                int NumLigne = -1;
                NumLigne = laTable_Table.getSelectedRow();
                // si aucune ligne sélectionnée
                if(NumLigne == -1){
                    dispose();
                    // on prépare les boutons
                    laFiche_client.setBtn_EnregistrerNonActif();
                    laFiche_client.setBtn_RechercherNonActif();
                    laFiche_client.getJTxTCode().setEditable(true);
                    // on rend les champs suivants non éditables
                    laFiche_client.getJTxTNom().setEditable(false);
                    laFiche_client.getJTxTType().setEditable(false);
                    // on affiche la fiche du client
                    laFiche_client.setVisible(true);
                }
                // si une ligne sélectionnée
                if(NumLigne != -1){
                    int choix = JOptionPane.showConfirmDialog(null, "Voulez-vous supprimer "
                        + "la fiche du client "
                        + laTable_Table.getValueAt(NumLigne, 0), "SUPPRESSION"
                        , JOptionPane.YES_NO_OPTION);
                    // 0 : oui 1 : non
                    if(choix == 0){
                        String vCode;
                        vCode = String.valueOf(laTable_Table.getValueAt(NumLigne, 0));
                        // on supprime l'enregistrement de la BD

```

```

        entite.Client leClient = new entite.Client(vCode);
        leClient.supprimerCRUD_Client(leClient);
        dispose();
        laFiche_client.setBtn_EnregistrerNonActif();
        laFiche_client.setBtn_RechercherNonActif();
        laFiche_client.getJTxTCode().setEditable(true);
        laFiche_client.getJTxTNom().setEditable(false);
        laFiche_client.getJTxTType().setEditable(false);
        laFiche_client.setVisible(true);
    }
}
});
}
return btn_Supprimer;
}

```

 Le code est assez long mais ne présente pas de difficultés particulières à ce stade de l'étude. Aussi, est-il directement et largement commenté pour l'essentiel. Il en sera ainsi pour la suite.

3. Modification d'un enregistrement

Plutôt que de créer une autre méthode, la méthode pour les ajouts va être à nouveau utilisée. On avait fait aussi le choix d'introduire une variante. Cette fois-ci, une ligne doit d'abord être sélectionnée avant d'être modifiée.

Dans la section Fenêtres Clients, la méthode **getBtn_Modifier()** de la classe **FenTableClient** n'avait volontairement pas été définie. Nous allons maintenant le faire en tenant compte de ce qui est mentionné plus haut. Il nous faut toutefois modifier auparavant la classe **FenFicheClient**. Celle-ci doit autoriser la modification de ses champs de saisie.


- Ouvrez la classe **FenFicheClient** et ajoutez les mutateurs suivants :

```

public void setJTxTCode(String vCode) {
    jTxTCode.setText(vCode) ;
}
public void setJTxTNom(String vNom) {
    jTxTNom.setText(vNom) ;
}
public void setJTxTType(String vType) {
    jTxTType.setText(vType);
}

public void setBtn_EnregistrerLibelle(String vLibelle) {
    btn_Enregistrer.setText(vLibelle);
}

```


 Un mutateur est également ajouté pour modifier le titre du bouton de sauvegarde en fonction du choix de l'utilisateur.

- Ouvrez maintenant la classe **GestionDemandes** et créez la méthode **demandeModifierClient()**.

```

public void demandeModifierClient(String vCode, String vNom, String vType){
    Client leClient = new Client(vCode, vNom, vType);
    leClient.modifierCRUD_Client(leClient);
}

```

 Pour la modification, c'est le premier constructeur qui est utilisé, le deuxième étant dédié aux recherches que nous verrons plus loin.

- Ouvrez la classe **FenTableClient** et importez de suite la classe **JOOptionPane**. Une boîte de dialogue est prévue dans les modifications.

```
import javax.swing.JOptionPane;
```

La méthode **getBtn_Modifier()** doit :

- identifier la ligne sélectionnée ;
- récupérer les valeurs de la ligne sélectionnée ;
- affecter ces valeurs aux champs de la fenêtre **FenFicheClient** qui présente les données en mode fiche ;
- rendre visible la fenêtre **FenFicheClient**.

■ Effectuez les modifications.

```
private JButton getBtn_Modifier() {
    if (btn_Modifier == null) {
        ...
        public void mouseClicked(java.awt.event.MouseEvent e) {
            boolean ok = false;
            // en cas d'oubli de sélection de lignes ou de table vierge
            try {
                int NumLigne = 0;
                NumLigne = laTable_Table.getSelectedRow();
                String vCodeClient;
                vCodeClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 0));
                // on affecte la valeur au champ code de la fenêtre de la fiche client
                laFiche_client.setJTxTCode(vCodeClient);
                // on interdit la modification du code
                // si vous l'autorisez, vérifiez que ce soit cohérent avec votre BD
                laFiche_client.getJTxTCode().setEditable(false);
                // on affecte la valeur au champ nom de la fenêtre de la fiche client
                String vNomClient;
                vNomClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 1));
                laFiche_client.setJTxTNom(vNomClient);
                // on affecte la valeur au champ type de la fenêtre de la fiche client
                String vTypeClient;
                vTypeClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 2));
                laFiche_client.setJTxTType(vTypeClient);
                ok = true;
            }
            catch (Exception oubliligne) {
                JOptionPane.showMessageDialog(null, "Sélectionnez auparavant" +
                    " la ligne à modifier"
                    + '\n'
                    + "ou effectuez un double clic sur la ligne",
                    "MODIFICATION", JOptionPane.INFORMATION_MESSAGE);
            }

            if(ok == true){
                dispose();
                laFiche_client.setVisible(true);
                // on désactive le bouton Supprimer de la fiche client
                laFiche_client.setBtn_SupprimerNonActif();
                // on désactive le bouton Chercher de la fiche client
                laFiche_client.setBtn_RechercherNonActif();
                // laFiche_client.getBtn_EnregistrerNonActif();
                laFiche_client.setBtn_EnregistrerLibelle("MODIFIER");
                // pour mettre le focus sur le champ nom
                laFiche_client.getJTxTNom().requestFocusInWindow();
            }
        }
    }
}
return btn_Modifier;
```



```
}
```



L'objet **laTable_Table** est un composant de type **JTable**. C'est une des propriétés de la fenêtre. Il utilise les méthodes suivantes :

- **getSelectedRow()** : retourne le numéro de la ligne sélectionnée sous la forme d'un entier.
- **getValueAt(int L, int C)** : retourne la valeur à l'intersection de la ligne et de colonne sous la forme d'un objet. La méthode **valueOf()** de la classe **String** permet de le convertir sous la forme d'une chaîne.

Un bloc **try/catch** est utilisé pour gérer l'oubli de la sélection d'une ligne.

Avant de tester l'application, des changements doivent être aussi apportés à la méthode **getBtn_Enregistrer()** de la classe **FenFicheClient**.

- Ouvrez la classe **FenFicheClient** et effectuez les modifications.

```
private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        ...
        public void actionPerformed(java.awt.event.ActionEvent e){
            String vCode = jTxTCode.getText();
            String vNom = jTxTNom.getText();
            String vType = jTxTType.getText();
            // action si seulement libellé du bouton est : SAUVEGARDER
            // --> création d'un client
            if (btn_Enregistrer.getText()=="SAUVEGARDER"){
                if(!vCode.equals("")){
                    // demande d'ajout d'un client dans la BD
                    // -----
                    leClientGestionClientBD.demandeEnregistrerClient(vCode, vNom, vType);
                    // si l'ajout s'est bien passée
                    // remettre à vide les champs de saisie
                    // pour une nouvelle saisie
                    jTxTCode.setText("");
                    jTxTNom.setText("");
                    jTxTType.setText("");
                    // on passe le focus au champ code
                    jTxTCode.requestFocusInWindow();
                }
                else
                {
                    JOptionPane.showMessageDialog(null, "La saisie du code client"
+ " est obligatoire",
                    "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
                }
            }

            // sinon nous sommes en mode modification
            // et le libellé du bouton est : MODIFIER
            // --> demande de modification des données d'un client
            else
            {
                leClientGestionClientBD.demandeModifierClient(vCode, vNom, vType);
            }
        }
    }
};
return btn_Enregistrer;
}
```



leClientGestionClientBD est une propriété de type **GestionDemandes** de la classe **FenFicheClient**.

- Testez l'application.

Les modifications ne sont pas prises en compte. Que nous manque-t-il ? Il reste en fait à définir la méthode habilitée à agir sur la base de données. Nous savons que c'est la classe **Client** qui possède ce type de méthode.

- Ouvrez la classe **Client** et définissez la méthode concernée.

```
// Modification d'un client dans la BD
// -----
public void modifierCRUD_Client(Client leClient){
    String requete = null;
    try {
        requete = "UPDATE client SET Nom = '" +leClient.getNom()+"'" + "," +
        "Type = '" +leClient.getType()+"'" +
        + " WHERE Code = '" +leClient.getCode()+"'";
        Statement state = laConnexion.createStatement();
        state.executeUpdate(requete);
        state.close();
        JOptionPane.showMessageDialog(null, "Modification réalisée",
        "MODIFICATION", JOptionPane.INFORMATION_MESSAGE);
    }
    catch (SQLException e){
        JOptionPane.showMessageDialog(null, "Modification non effectuée."
        + " Ce code client n'existe pas.",
        "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        System.out.println("Modification non effectuée");
    }
}
```

- Testez l'application.

La méthode **getBtn_Enregistrer()** sert maintenant à la fois pour sauvegarder la création d'un nouveau client et les modifications d'un client existant dans la base de données MySQL.

4. Gestion du double clic sur un JTable

Pour rendre les modifications plus aisées, nous gérons le double clic sur les lignes du composant **JTable**. Cette action ouvre la fenêtre **FenFicheClient** avec les données de la ligne sélectionnée présentées en mode fiche.


- Ouvrez la classe **FenTableClient** et redéfinissez la méthode **getJTable()**.

```
private JTable getJTable() {
    if (laTable_Table == null) {
        laTable_Table = new JTable();
        // la table est remplie
        laTable_Table = remplirTable();
        laTable_Table.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(java.awt.event.MouseEvent e) {
                // gestion du double clic sur une ligne de la table avec un simple comptage
                if (e.getClickCount() == 2) {
                    int NumLigne = 0;
                    NumLigne = laTable_Table.getSelectedRow();
                    String vCodeClient;
                    vCodeClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 0));
                    // on affecte la valeur au champ code de la fenêtre de la fiche client
                    laFiche_client.setJTxtCode(vCodeClient);
                    // on interdit la modification du code
                    // si vous l'autorisez, vérifiez que ce soit cohérent avec votre BD
                    laFiche_client.getJTxtCode().setEditable(false);
                    // on affecte la valeur au champ nom de la fenêtre de la fiche client
                    String vNomClient;
                    vNomClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 1));
                    laFiche_client.setJTxtNom(vNomClient);
                }
            }
        });
    }
}
```

```

        // on affecte la valeur au champ type de la fenêtre de la fiche client
        String vTypeClient;
        vTypeClient = String.valueOf(laTable_Table.getValueAt(NumLigne, 2));
        laFiche_client.setJTxTType(vTypeClient);
        dispose();
        laFiche_client.setVisible(true);
        // on désactive le bouton Supprimer de la fiche client
        laFiche_client.setBtn_SupprimerNonActif();
        // on désactive le bouton Chercher de la fiche client
        laFiche_client.setBtn_RechercherNonActif();
        // laFiche_client.getBtn_EnregistrerNonActif();
        laFiche_client.setBtn_EnregistrerLibelle("MODIFIER");
        // pour mettre le focus sur le champ nom
        laFiche_client.getJTxTNom().requestFocusInWindow();
    }
}
});
}
return laTable_Table;
}

```

 La gestion du double clic est très simple, comme vous pouvez le voir. Vous constatez qu'il existe de nombreuses lignes de code communes avec la méthode **getBtn_Modifier()**. Il conviendrait d'écrire une méthode regroupant ces lignes. Vous disposez désormais de toutes les connaissances pour le faire seul. Reportez-vous si nécessaire au chapitre Gestion de la connexion et voyez la méthode **contrôleConnexion_Appel()**.

La méthode **remplirTable()** a été réalisée au chapitre Gestion de l'affichage des données.

5. Recherche d'enregistrements

Nous arrivons au terme de notre projet. Il nous reste une dernière fonctionnalité à réaliser : la recherche d'enregistrements.

Le bouton **RECHERCHER** de la fenêtre **FenTableClient** est déjà opérationnel. Pour faire simple, cette recherche étant uniquement basée sur le code, il ne faut pas oublier de rendre les champs de saisie du nom et du type non éditables.

- Ouvrez la classe **FenTableClient** et insérez les lignes de code.

```

private JButton getBtn_Rechercher() {
    if (btn_Rechercher == null) {
        ...
        public void mouseClicked(java.awt.event.MouseEvent e) {
            ...
            laFiche_client.getJTxTNom().setEditable(false);
            laFiche_client.getJTxTType().setEditable(false);
            ...
        }
    }
});
return btn_Rechercher;
}

```

Nous n'avons pas comme pour les autres fonctionnalités à redéfinir dans la classe **Client** la méthode de recherche. Nous l'avons déjà fait pour gérer l'affichage des données dans le **JTable** (cf. chapitre Gestion des traitements). Par contre, il nous faut personnaliser auparavant la fenêtre **FenResultatRecherche()** qui affiche le résultat de la recherche en mode table. Cette fenêtre comporte aussi un composant **JTable**. Voici la partie de code le concernant :

```

// Initialisation du composant JTable
private JTable getJTable() {
    if (laTable_Table == null) {
        laTable_Table = new JTable();
        laTable_Table = remplirTable();
    }
    return laTable_Table;
}

```

```
private JTable remplirTable(){
    Client lesClients = new Client(leCode);
    lesClients.chercherCRUD_Clients(lesClients);

    Vector<Vector> Lignes = lesClients.getTabLignes();
    Vector<String> NomColonnes = lesClients.getNomColonnes();

    return new JTable(Lignes,NomColonnes);
}
```

Si vous éprouvez des difficultés à comprendre ces deux méthodes, reportez-vous au chapitre Gestion de l’affichage de données.



Le code complet de la classe **FenResultatRecherche()** est donné en fin de ce chapitre.

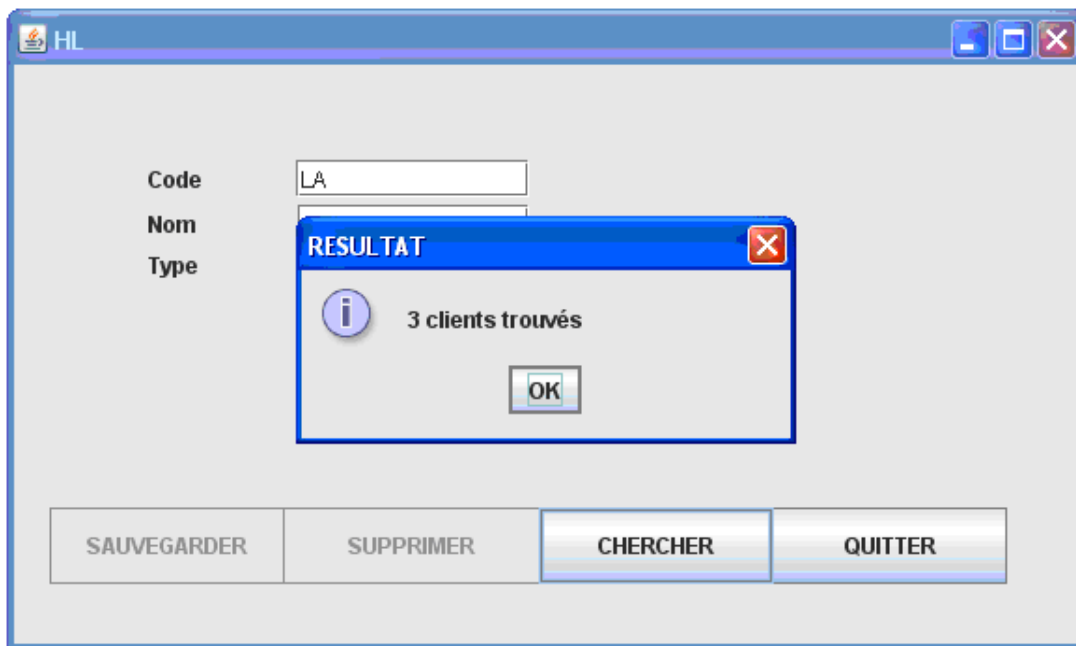
- Ouvrez maintenant la classe **GestionDemandes** et créez la méthode **demandeChercherClient()**.

```
public int demandeChercherClient(String vCode){
    Client leClient = new Client(vCode);
    return leClient.chercherCRUD_Clients(leClient);
}
```



C’est la méthode **chercherCRUD_Clients()** qui en définitive effectue la recherche dans la base de données.

L’utilisateur est informé du nombre de clients trouvés avant affichage. Un contrôle s’effectue ici uniquement sur la saisie du code. Exemple :



Code	Nom	Type
LA1	LANGLE	Artisan
LA2	LAUMET	Artisan
LA3	LARUE	Particulier

- Pour finir, ouvrez la classe **FenFicheClient** et redéfinissez la méthode **getBtn_Rechercher ()**.

```
private JButton getBtn_Rechercher() {
    if (btn_Rechercher == null) {
        btn_Rechercher = new JButton();
    }
}
```

```


btn_Rechercher.setBounds(new Rectangle(281, 237, 126, 41));
btn_Rechercher.setText("CHERCHER");
btn_Rechercher.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent e) {
        String vCode = jTxTCode.getText();
        String vNom = jTxTNom.getText();
        String vType = jTxTType.getText();
        if(!vCode.equals("") || !vNom.equals("") || !vType.equals("")){
            // pour faire simple, recherche ici uniquement à partir du code
            int nbEnreg = leClientGestionClientBD.demandeChercherClient(vCode);
            if(nbEnreg > 0){
                switch(nbEnreg)
                {
                    case 0:
                        JOptionPane.showMessageDialog(null, "Aucun client trouvé."
                            + " Ce code client n'existe pas.",
                            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
                        break;
                    case 1:
                        JOptionPane.showMessageDialog(null, nbEnreg + " client trouvé",
                            "RESULTAT", JOptionPane.INFORMATION_MESSAGE);
                        break;
                    default:
                        JOptionPane.showMessageDialog(null, nbEnreg + " clients trouvés",
                            "RESULTAT", JOptionPane.INFORMATION_MESSAGE);
                }
                btn_Enregistrer.setEnabled(true);
                jTxTCode.setText(vCode);
                new FenResultatRecherche(vCode);
            }
            if(nbEnreg == 0){
                JOptionPane.showMessageDialog(null, "Aucun client trouvé."
                    + " Ce code client n'existe pas.",
                    "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
            }
        }
        else
        {
            JOptionPane.showMessageDialog(null, "Saisissez au moins"
                + " un critère de recherche",
                "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        }
    }
});
}
return btn_Rechercher;
}

```

- Faites un test sur une égalité stricte du code puis après avoir ajouté plusieurs enregistrements dont le code commence par les mêmes lettres.

Pour améliorer cette recherche, il existe au moins deux possibilités :

- modifier la méthode actuelle pour qu'elle prenne en compte également les autres champs ;
- créer une nouvelle méthode avec la requête SQL appropriée.

 La nouvelle méthode peut porter le même nom mais doit différer par le nombre de paramètres ou par le type des paramètres.

Classe **FenResultatRecherche()**

```
package dialogue;
```

```

import java.awt.Font;
import java.awt.Rectangle;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

import java.util.Vector;

import entite.Client;

public class FenResultatRecherche extends JFrame {
    private static final long serialVersionUID = 1L;

    // propriétés non graphiques de la fenêtre
    private String leCode;
    // mutateur
    public void setCode(String code){
        leCode = code;
    }
    // accesseur
    public String getleCode(){
        return leCode;
    }

    // propriétés graphiques de la fenêtre
    private JPanel jContentPane = null;
    private JScrollPane jScrollPane = null;
    private JTable laTable_Table = null;
    private JLabel titre_Label = null;
    private JButton btn_Quitter = null;

    public FenResultatRecherche(String code) {
        super();
        // pour faciliter la saisie
        leCode = code + "%";
        System.out.println("lors création fiche client : "+leCode);
        initialize();
    }

    private void initialize() {
        this.setSize(631, 360);
        this.setContentPane(getJContentPane());
        this.setTitle("HL");
        ImageIcon image = new ImageIcon("images\\article.gif");
        this.setIconImage(image.getImage());
        this.setLocationRelativeTo(null); // On centre la fenêtre sur l'écran
        this.setVisible(true);
    }

    private JPanel getJContentPane() {
        if (jContentPane == null) {
            titre_Label = new JLabel();
            titre_Label.setBounds(new Rectangle(73, 9, 315, 16));
            titre_Label.setFont(new Font("Comic Sans MS", Font.BOLD, 18));

            titre_Label.setText("CLIENTS");
            jContentPane = new JPanel();
            jContentPane.setLayout(null);
            jContentPane.add(jScrollPane, null);
            jContentPane.add(titre_Label, null);
            jContentPane.add(btn_Quitter, null);
        }
    }

```

```

        return jContentPane;
    }

    private JScrollPane getJScrollPane() {
        if (jScrollPane == null) {
            jScrollPane = new JScrollPane();
            jScrollPane.setBounds(new Rectangle(75, 33, 453, 200));
            jScrollPane.setViewportView(getJTable());
        }
        return jScrollPane;
    }

    // Initialisation du composant JTable
    private JTable getJTable() {
        if (laTable_Table == null) {
            laTable_Table = new JTable();
            // voir la méthode remplirTable() plus loin
            laTable_Table = remplirTable();
        }
        return laTable_Table;
    }

    // pour afficher les enregistrements de la table client dans le composant table
    private JTable remplirTable(){
        Client lesClients = new Client(leCode);
        lesClients.chercherCRUD_Clients(lesClients);
        Vector<Vector> Lignes = lesClients.getTabLignes();
        Vector<String> NomColonnes = lesClients.getNomColonnes();
        return new JTable(Lignes,NomColonnes);
    }

    private JButton getBtn_Quitter() {
        if (btn_Quitter == null) {
            btn_Quitter = new JButton();
            btn_Quitter.setBounds(new Rectangle(466, 259, 115, 44));
            btn_Quitter.setFont(new Font("Comic Sans MS", Font.BOLD, 12));
            btn_Quitter.setText("QUITTER");
            btn_Quitter.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    dispose();
                }
            });
        }
        return btn_Quitter;
    }
}

```

Création du jar

L'application telle qu'elle avait été définie est terminée. Pour qu'elle devienne autonome, c'est-à-dire exécutable sans l'aide d'Eclipse, nous allons la transformer en une archive **Jar**, Java Archive. Il s'agit d'un fichier compressé contenant tous les fichiers indispensables à l'exécution de l'application notamment les fichiers **class** et le fichier **manifest**.

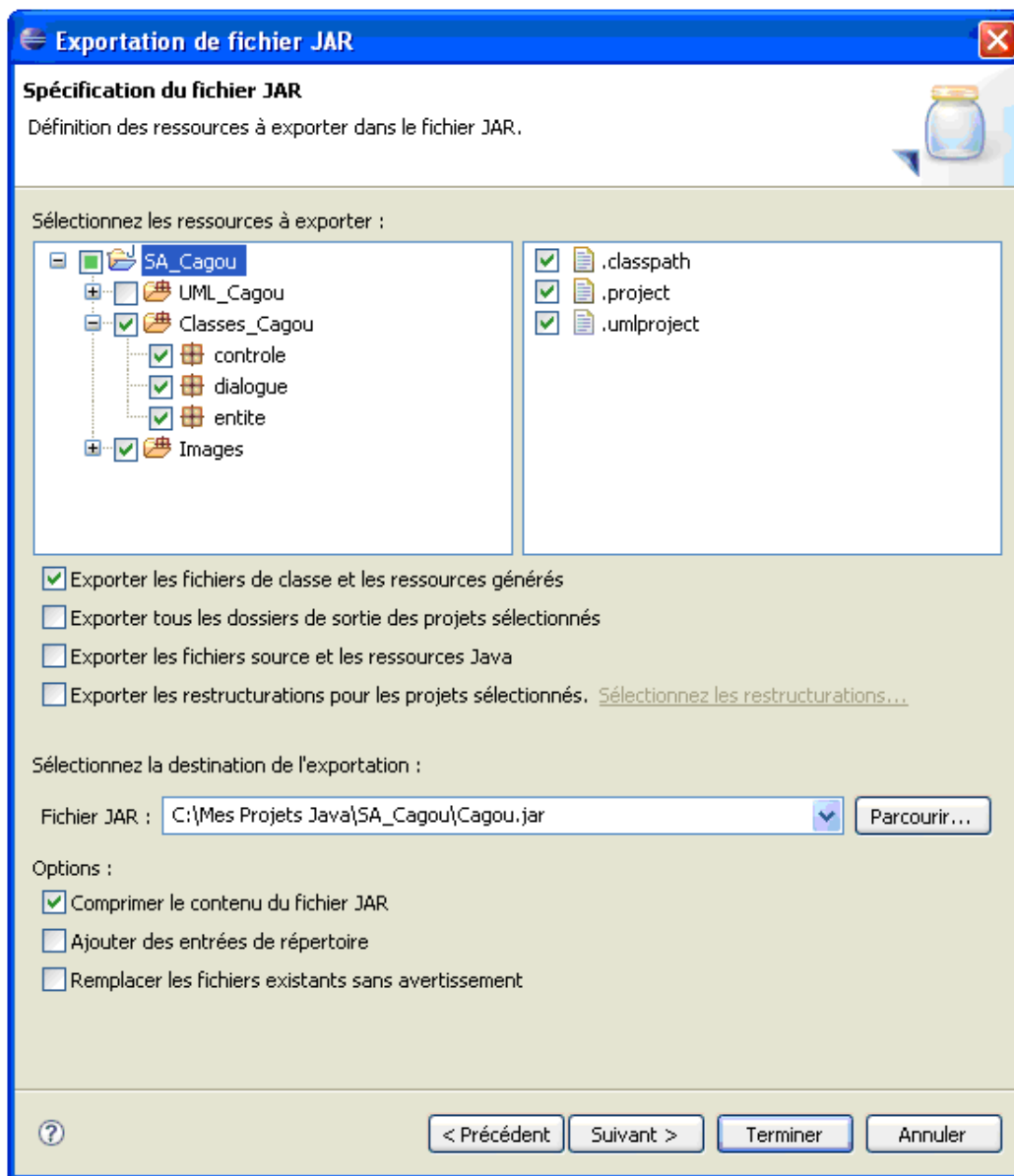
Les fichiers class sont vos fichiers Java que vous avez créés sous Eclipse et qui ont été compilés en byte-code pour fonctionner avec la **JVM**, la machine virtuelle Java. Le fichier manifest comporte des informations comme le nom de la classe principale qui contient la méthode **main()** ou les paquetages qui ont été scellés.

Contenu du fichier manifest du Jar Cagou où tous les paquetages sont scellés :

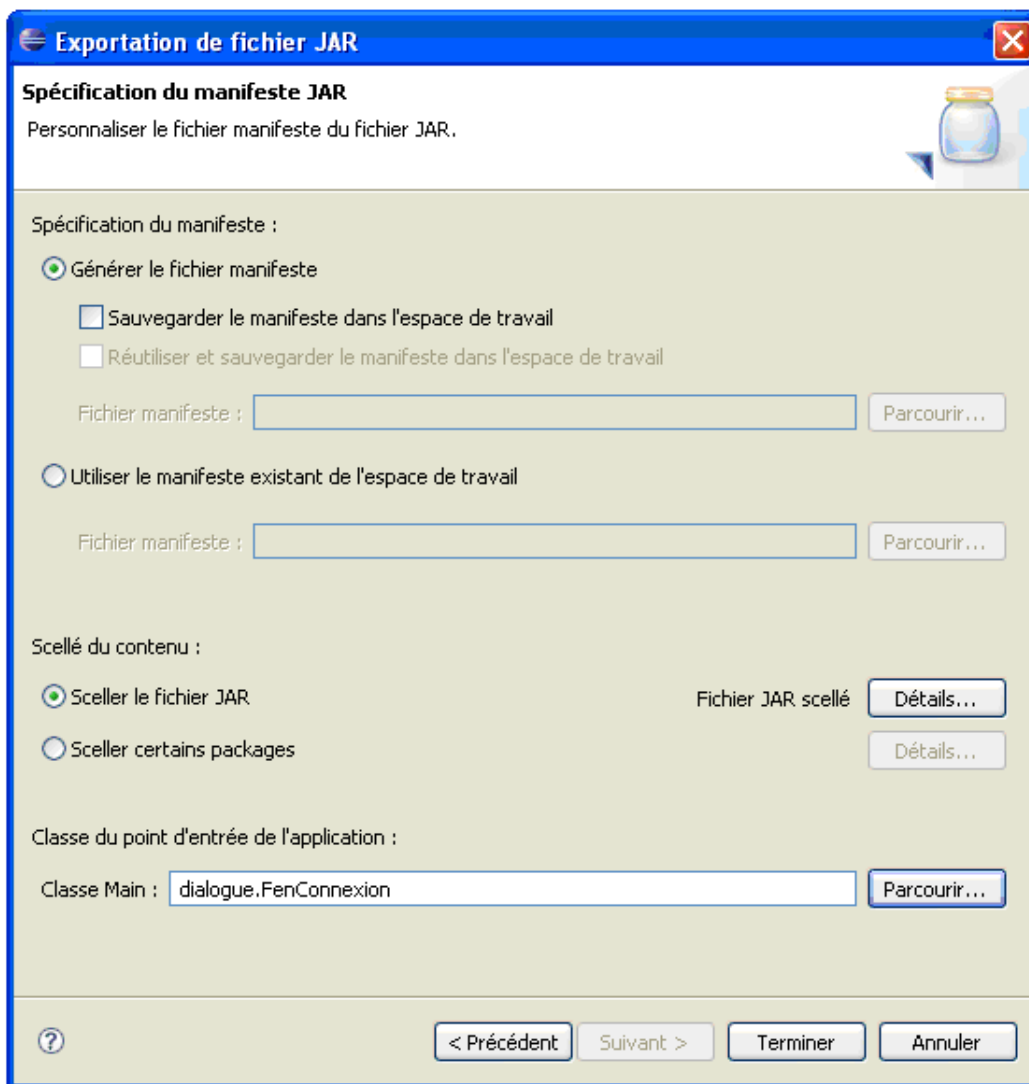
```
Manifest-Version: 1.0
Sealed: true
Main-Class: dialogue.FenConnexion
```

Passons à la création du Jar. Sous Windows et bien sûr sous Linux, comme pour une application Java complète tout peut être fait à partir de lignes de commandes sans IDE. Nous utilisons Eclipse une fois de plus pour gagner du temps.

- Effectuez un clic droit sur le projet et choisissez **Exporter**. Par défaut l'option **Fichier Jar** est déjà sélectionnée. Cliquez sur **Suivant >**.
- Vous pouvez décocher le paquetage UML. Précisez le dossier de destination, ici le dossier du projet. Vérifiez que le dossier que le dossier **Images** est bien coché, laissez les options principales et passez à la fenêtre suivante.



- Gardez les options par défaut pour la fenêtre suivante. Cliquez sur **Suivant >**.
- Conservez les options proposées et indiquez la classe principale (celle qui contient la méthode **main()**). Cliquez sur **Terminer**.



- Testez l'application autonome en cliquant sur son icône.

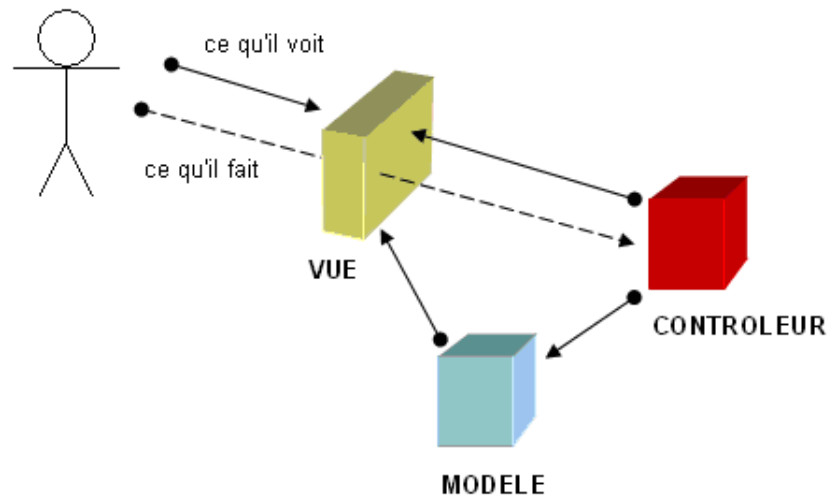


Présentation

Le projet SA_Cagou a été bâti en prenant soin dès la phase d'analyse de séparer les classes selon leur type : dialogue, contrôle et entité. Le **modèle MVC** ou design pattern **Model View Controller** formalise cette séparation et vise en outre à la synchronisation des vues (classes dialogue) et des modèles (classes entité) en s'appuyant sur les contrôleurs (classes contrôle).

Dans un contexte MVC, les données constituent les modèles, les classes graphiques les vues. Modèles et vues sont totalement indépendants les uns des autres. Nous dirons qu'ils ne se connaissent pas. Comment alors les actions de l'utilisateur via les vues peuvent-ils modifier les modèles et inversement, comment les données attendues peuvent-ils être affichées ? C'est justement les contrôleurs qui vont établir le lien.

Autre particularité importante. Le modèle, s'il ne connaît pas les vues qui affichent ses données, a néanmoins la possibilité de notifier à celles-ci tout changement le concernant.



Le schéma nous montre que l'utilisateur n'a aucune connaissance du contrôleur. C'est pourtant lui qui transmet de manière transparente pour l'utilisateur les requêtes au modèle pour traitement. Les vues sont mises à jour soit par le contrôleur soit par le modèle par un système de notification basé sur la notion d'événements et d'écouteurs d'événements.

Jusqu'à présent dans notre projet, pour effectuer une mise à jour de la fenêtre présentant les données en mode table, il faut la refermer, ouvrir une autre fenêtre pour effectuer les traitements CRUD puis rouvrir la première fenêtre. Une nouvelle ouverture correspond en fait à une nouvelle instance de la fenêtre en mode table avec interrogation de la base et mise à jour du composant JTable (revoir au besoin les sections Gestion de l'affichage des données et Gestion des traitements du chapitre Développement).

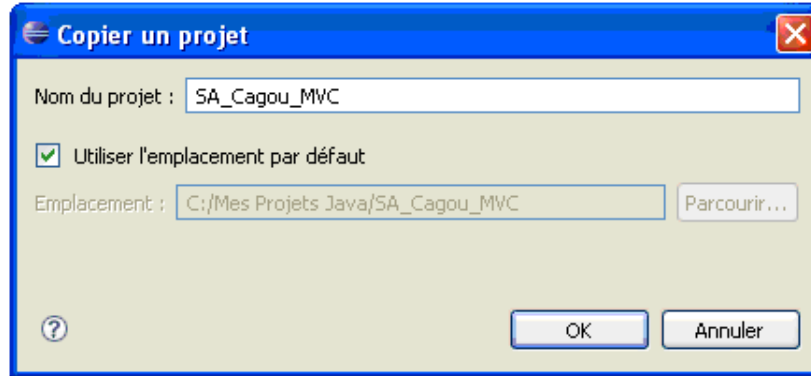
Ce procédé fonctionne mais ne permet pas de réaliser une mise à jour immédiate de la fenêtre en mode table. Il devient alors impossible de mettre à jour de manière simultanée différentes vues des mêmes données suite à des modifications les concernant.

Dans ce chapitre, nous voyons donc comment mettre en œuvre le modèle MVC avec le composant swing JTable pour nous affranchir de cette limite.

Création du modèle

Nous allons auparavant créer une copie du projet.


- Effectuez un clic droit sur le projet SA_Cagou dans l'explorateur de packages puis choisissez **Copier**.
- Puis toujours dans l'explorateur de packages effectuez un autre clic droit et renommez le projet "SA_Cagou_MVC".



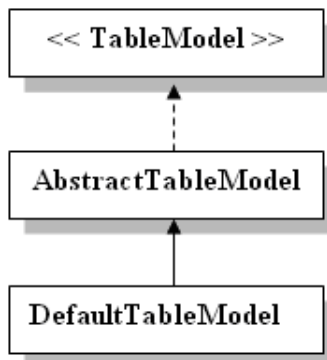
Pour rendre les vues totalement indépendantes des données, il faut commencer par créer un modèle. La classe **Client** va être modifiée pour permettre la création du modèle.


- Ouvrez cette classe dans le paquetage **entite** et procédez à l'importation suivante :

```
import javax.swing.table.DefaultTableModel;
```

 La classe **DefaultTableModel** étend la classe **AbstractTableModel** qui elle-même implémente l'interface **TableModel**.

- L'interface **TableModel** spécifie des méthodes utiles pour gérer les données tabulaires contenues par un modèle (`getColumnName(int columnIndex)`, `getRowCount()`, ...)
- La classe **AbstractTableModel** redéfinit la plupart des méthodes de l'interface **TableModel** en proposant un code par défaut. Elle fournit en outre les méthodes pour l'enregistrement des vues auprès du modèle et celles pour la notification des événements survenus.
- La classe **DefaultTableModel** est une sous-classe de la classe **AbstractTableModel**. Elle peut donc utiliser et redéfinir si nécessaire les méthodes de cette dernière.



 La consultation de l'aide de SUN s'avère indispensable. Lancez celle-ci puis choisissez dans le volet **Packages**, le paquetage **javax.swing.table**.



- Ajoutez la propriété suivante :

```
private DefaultTableModel leModele = new DefaultTableModel();
```

- Ajoutez un accesseur pour cette propriété.

```
public DefaultTableModel getLeModeleClients(){
    return leModele;
}
```




C'est ce modèle qui sera attaché au composant JTable.

Actuellement dans le projet, lors de l'instanciation de la classe **FenTableClient**, une sélection de tous les enregistrements de la table **Client** est effectuée. Il faut donc intervenir au niveau du code effectuant cette opération dans la classe **Client**.

- Modifiez la méthode **chercherCRUD_Clients(Client lesClients)**.

```
public int chercherCRUD_Clients(Client_MVC lesClients){
    ...
    for( i=1; i <= infojeuEnregistrements.getColumnCount(); i++){
        // les titres des colonnes sont récupérés avec la méthode getColumnLabel
        String nomColonneModel = infojeuEnregistrements.getColumnLabel(i);
        // puis ajoutés au modèle
        // -----
        leModele.addColumn(nomColonneModel);
    }
    while(jeuEnregistrements.next()) {
        Vector<String> ligne = new Vector<String>();
        for(i=1; i <= infojeuEnregistrements.getColumnCount(); i++) {
            String chaine_champ = jeuEnregistrements.getString(i);
```

```
        ligne.add(chaine_champ);
    }
    nbClients = nbClients + 1;
    // le vecteur ligne est ajouté au modèle
    // -----
    leModele.addRow(ligne);
}
...
}
```

 Après l'exécution de la méthode, le modèle contient les données et les titres des colonnes.

- Lancez l'application.


Plus aucune donnée n'est visible mais ceci est tout à fait normal puisque l'affichage est assuré par le composant JTable et non par le modèle.

Création de la vue

Pour que les données soient visibles, il suffit de préciser quel modèle doit être affiché par le composant JTable.

- Ouvrez la classe **FenTableClient** dans le paquetage **dialogue** et procédez aux importations suivantes.

```
import javax.swing.event.TableModelListener;  
import javax.swing.event.TableModelEvent;
```

 Toute vue sur un modèle doit implanter l'interface **TableModelListener**. Cette interface permet à la vue d'être à l'écoute des événements modifiant l'état du modèle. Elle ne dispose que d'une seule méthode **tableChanged(TableModelEvent e)** qui est redéfinie en fonction des événements survenus.


- Implémentez l'interface **TableModelListener**.

```
public class FenTableClient extends JFrame implements TableModelListener {
```

- Eclipse vous signale que la méthode **tableChanged(TableModelEvent e)** doit être implémentée. Double cliquez sur **Ajouter des méthodes non implémentées**.

```
public class FenTableClient extends JFrame implements TableModelListener {  
    Ajouter des méthodes non implémentées  
    Associer le type 'FenTableClient' à la valeur abstract  
    Renommer dans le fichier (accès direct par Ctrl+2, R)  
    1 méthode(s) à implémenter :  
    - javax.swing.event.TableModelListener.tableChanged()
```


```
public void tableChanged(TableModelEvent arg0) {  
    // TODO Raccord de méthode auto-généré  
}
```

 Pour chaque type d'événement, il est possible de programmer une action particulière. La mise à jour de la vue avec le composant JTable est par contre automatique. Notez le type du paramètre. Il s'agit de la classe **TableModelEvent** qui étend la classe **EventObject**.

```
java.lang.Object  
└─ java.util.EventObject  
    └─ javax.swing.event.TableModelEvent
```

- Ajoutez la propriété suivante :

```
private DefaultTableModel leModele;
```

 La propriété **leModele** est initialisée lors de l'exécution de la méthode **remplirTable()**.

Il faut maintenant intervenir sur le code de cette méthode à l'origine de l'appel des méthodes permettant de sélectionner les enregistrements de la table **Client**.

- Modifiez la méthode **remplirTable()**.

```
private JTable remplirTable(){  
    String vCode = "";  
    entite.Client lesClients = new entite.Client (vCode);  
    lesClients.chercherCRUD_Clients(lesClients);
```

```

leModele = lesClients.getLeModeleClients();

laTable_Table = new JTable(leModele);

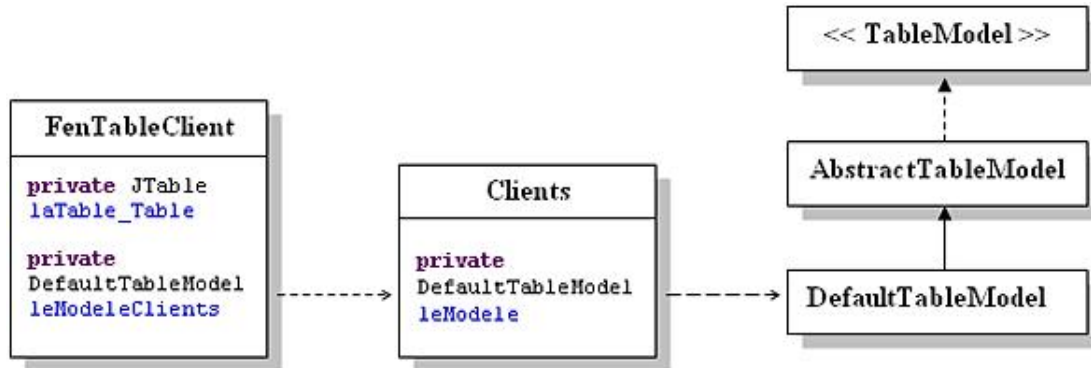
return laTable_Table;
}


```

 La table est retournée avec le modèle.

- Testez l'application.

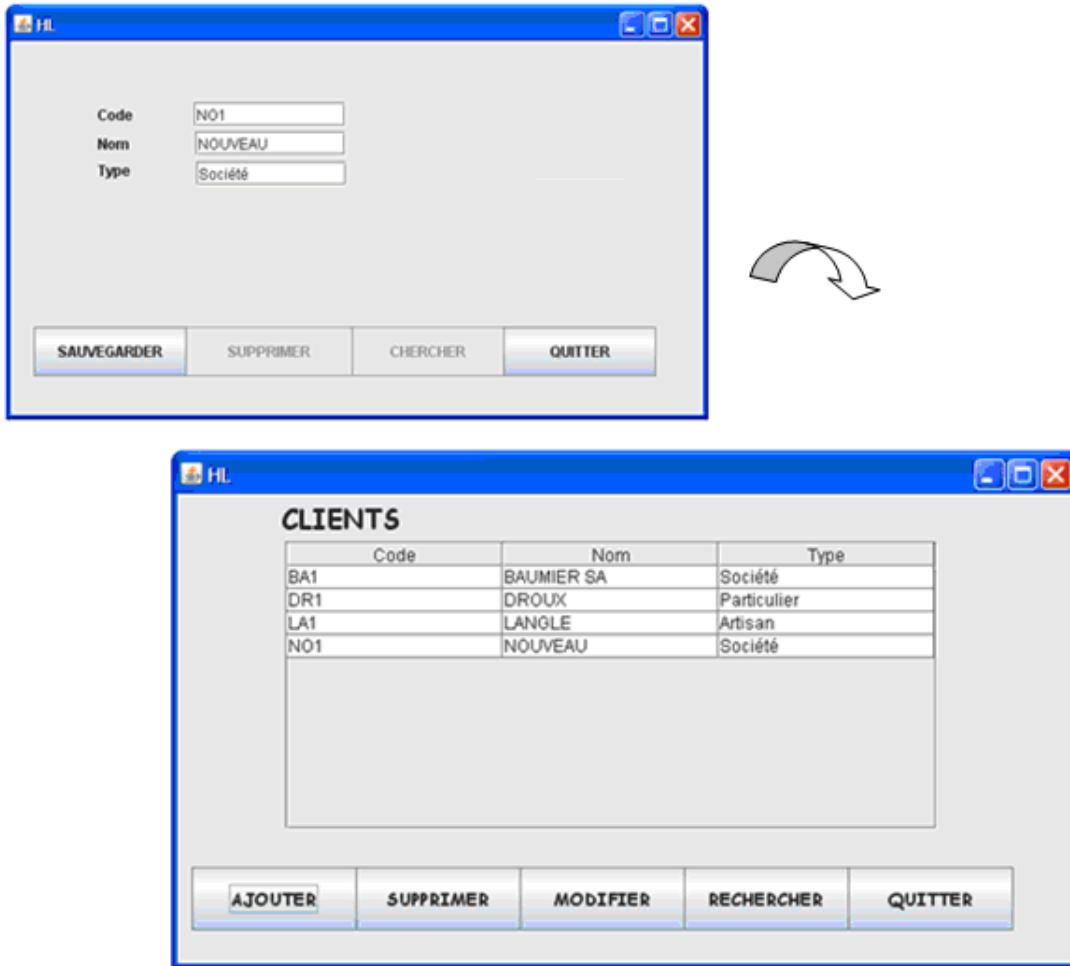
Le composant JTable affiche les données du modèle. Le schéma ci-après rappelle les principales relations entre les classes.



 Avec le composant JTable, la vue et le contrôleur ne sont pas dissociés. Il n'est donc pas utile de créer une nouvelle classe de type contrôleur.

Mise à jour de la fenêtre en mode table

Nous allons mettre en œuvre concrètement le modèle MVC en créant un nouveau client à partir de la fenêtre en mode fiche. Cette création doit être visible dans la fenêtre en mode table qui pour l'occasion restera ouverte.



- Dans le code de la classe **FenTableClient**, recherchez la méthode **getBtn_Ajouter()** puis désactivez ou supprimez la ligne qui referme la fenêtre.

```
// FenTableClient.this.dispose(); // à désactiver ou à supprimer
```

- Dans le code de la classe **FenFicheClient**, recherchez la méthode **getBtn_Quitter()** puis désactivez ou supprimez les lignes qui créent une nouvelle instance de **FenTableClient** et qui la rendent visible.

```
// FenTableClient laTable_Client = new FenTableClient(); // à désactiver ou à supprimer  
// laTable_Client.setVisible(true); // à désactiver ou à supprimer
```

- Lancez l'application. Testez le bouton **AJOUTER** de la fenêtre en mode table.

La fenêtre d'ajout en mode fiche apparaît et celle en mode table reste également visible.

- Déplacez vos fenêtres de sorte que vous puissiez voir si l'ajout à partir de la fiche est répercuté dans la table. Ajoutez le client suivant :

Code	<input type="text" value="NO1"/>
Nom	<input type="text" value="NOUVEAU"/>
Type	<input type="text" value="Société"/>

À ce stade, l'ajout d'un nouveau client est bien pris en compte et l'enregistrement correspondant est créé dans la base de données. Par contre, la mise à jour simultanée de la table n'est pas encore effective.

Pour que la mise à jour simultanée devienne effective, plusieurs conditions doivent être réalisées :

- l'ajout doit concerner le même modèle que celui utilisé par la table ;
 - le constructeur de la classe **FenFicheClient** doit être modifié en lui ajoutant un paramètre de type **DefaultTableModel** ;
 - le modèle doit être agrandi d'une ligne, en l'occurrence un vecteur qui contiendra les nouvelles données ;
 - le modèle modifié doit pouvoir notifier aux vues les changements le concernant.
- Ouvrez la classe **FenFicheClient** et procédez à l'importation suivante :

```
import javax.swing.table.DefaultTableModel;
```

- Ajoutez une nouvelle propriété.

```
private DefaultTableModel leModeleTransmisParFenTable;
```

- Modifiez le constructeur de la classe.

```
public FenFicheClient(DefaultTableModel leModele) {
    super();
    initialize();
    leModeleTransmisParFenTable = leModele;
}
```



Les objets des classes **FenFicheClient** et **FenTableClient** sont ainsi des vues sur le même modèle.

Il reste à modifier la méthode **getBtn_Enregistrer()**. Celle-ci doit ajouter un vecteur au modèle avec les données du nouveau client.

- Modifiez la méthode **getBtn_Enregistrer()**.

```
private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        ...
        public void actionPerformed(java.awt.event.ActionEvent e){
            ...
            if (btn_Enregistrer.getText()=="SAUVEGARDER"){if(!vCode.equals("")){
                // demande d'ajout d'un client dans la BD
                // -----
                leClientGestionClientBD.demandeEnregistrerClient(vCode, vNom,
vType);
                ...
                jTxTCode.requestFocusInWindow();

                // ajout
                Vector<String> ligne = new Vector<String>();
                leModeleTransmisParFenTable.addRow(ligne);
```

```

        int nbLignes = leModeleTransmisParFenTable.getRowCount();
        nbLignes = nbLignes - 1;
        leModeleTransmisParFenTable.setValueAt(vCode, nbLignes, 0);
        leModeleTransmisParFenTable.setValueAt(vNom, nbLignes, 1);
        leModeleTransmisParFenTable.setValueAt(vType, nbLignes, 2);
        // fin ajout
    }
    else {
        JOptionPane.showMessageDialog(null, "La saisie du code client"
            + " est obligatoire",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
    }
}
else
{
    // demande de modification d'un client dans la BD
    leClientGestionClientBD.demandeModifierClient(vCode, vNom, vType);
}
}
});
}
return btn_Enregistrer;
}

```

Avant de procéder à un nouveau test, il faut prendre en compte la modification du constructeur et apporter quelques modifications au code de la classe **FenTableClient**.

- Ouvrez la classe **FenTableClient** et supprimez l'affectation à la déclaration de la propriété **laFiche_client**.

```
private FenFicheClient laFiche_client;
```

- Modifiez la méthode **remplirTable()**.


```

private JTable remplirTable(){
    String vCode = "";
    entite.Client lesClients = new entite.Client (vCode);
    // demande de recherche dans la base de données
    lesClients.chercherCRUD_Clients(lesClients);
    // 1
    leModeleClients = lesClients.getLeModeleClients();

    // 2
    laFiche_client = new FenFicheClient(leModeleClients);

    // 3
    laTable_Table = new JTable(leModeleClients);
    return laTable_Table;
}

```

 La modification consiste à l'insertion d'une seule ligne de code. En 1, le modèle de la table est récupéré. En 2 (nouvelle ligne insérée), ce modèle est passé en paramètre à un objet de la classe **FenFicheClient**. En 3, la table est retournée avec son modèle pour affichage dans la fenêtre en mode table.

La mise à jour simultanée de la table est désormais possible.

- Testez l'application.

Nous allons effectuer maintenant les modifications de code pour la suppression d'un client.

- Toujours dans la classe **FenTableClient** où s'effectue la suppression suite à un double clic, modifiez la méthode **getBtn_Supprimer()**.

```

private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        ...
        btn_Supprimer.addActionListener
(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                ...
                if(NumLigne == -1){
                    ...
                }
                if(NumLigne != -1){
                    ...
                    if(choix == 0){
                        ...

                        // suppression du modèle sur double-clic
                        leModeleClients.removeRow(NumLigne);

                    }
                }
            }
        });
    }
    return btn_Supprimer;
}

```



Le modèle a déjà été récupéré avec la méthode **remplirTable()**.

Dans le projet SA_Cagou, il est également possible de supprimer un client en mode fiche avec la classe **FenFicheClient**. Pour effectuer cette suppression dans le contexte MVC, il faut :

- transmettre le modèle à l'instance la classe **FenFicheClient** depuis l'instance de la classe **FenTableClient** ;
- parcourir le modèle ;
- comparer le code du client à supprimer avec la valeur contenue dans la cellule de la première colonne de chaque ligne avec la méthode **getValueAt(int row, int column)** de la classe **DefaultTableModel** ;
- supprimer la ligne du modèle avec la méthode **removeRow(int row)** de la classe **DefaultTableModel**.



Pour plus d'informations sur la classe **DefaultTableModel** consulter l'aide de Sun.

```

java.lang.Object
├─ javax.swing.table.AbstractTableModel
├─ javax.swing.table.DefaultTableModel

```

- Testez l'application.

Mise à jour simultanée de plusieurs fenêtres

1. Ajout d'un enregistrement

Nous n'avons vu qu'une facette du modèle MVC. Il est possible de réaliser des opérations de synchronisation plus complexes en utilisant les méthodes pour la notification et la détection des événements.

Nous allons créer une fenêtre identique à **FenTableClient**. Pour faire simple, elle ne différera de cette dernière uniquement que par la couleur et sera nommée **FenTableN2Client**. Lors de l'ajout ou de la suppression d'un client, trois fenêtres seront désormais visibles :

- la fenêtre **FenFicheClient** qui pilote l'ajout ou la suppression ;
- les fenêtres **FenTableClient** et **FenTableN2Client** qui se mettent à jour suite à une notification du modèle.

Trois opérations sont nécessaires :

- l'enregistrement de la vue dans la liste des écouteurs du modèle ;
- la notification par le modèle aux vues de l'événement à l'origine de la modification de son état avec une méthode du type **fireXXX()** - la méthode **fireTableDataChanged()** est prise en charge par défaut avec le composant JTable ;
- les actions menées par les vues en fonction de l'événement survenu.

Commençons par enregistrer la vue **FenTableClient** auprès du modèle.

- Ouvrez la classe **FenTableClient**. Ajoutez le code pour l'enregistrement à la méthode **remplirTable()**.

```
private JTable remplirTable(){
    String vCode = "";
    entite.Client lesClients = new entite.Client (vCode);
    lesClients.chercherCRUD_Clients(lesClients);
    leModeleClients = lesClients.getLeModeleClients();

    // ENREGISTREMENT MVC
    leModeleClients.addTableModelListener(this);

    laFiche_client = new FenFicheClient(leModeleClients);
    laTable_Table = new JTable(leModeleClients);
    return laTable_Table;
}
```

 La méthode **addTableModelListener(TableModelListener l)** est héritée de la classe abstraite **AbstractTableModel**. Cette vue peut désormais écouter toute modification du modèle qui lui sera notifiée par une méthode du modèle de type **fireXXX()**.

- Modifiez la méthode **tableChanged (...)**. Pour l'instant, les actions sont de simples messages.

```
public void tableChanged(TableModelEvent unEvenement) {
    switch(unEvenement.getType()){
        case TableModelEvent.INSERT:
            System.out.println("Je me suis mis à jour, il y a eu une
insertion.");
            break ;
        case TableModelEvent.DELETE:
            System.out.println("Je me suis mis à jour, il y a eu une
suppression.");
    }
```

```

        break;
    }
}

```



La vue dès qu'elle est notifiée, en réponse à un **fireXXX()** du modèle exécute la méthode **tableChanged (TableModelEvent e)**. Celle-ci contient des actions ou autres méthodes écrites en fonction du type d'événement à l'origine des changements notifiés (déclarés) par le modèle. Cette méthode est l'unique méthode de l'interface **TableModelListener()** présente dans le paquetage **javax.swing.event**.

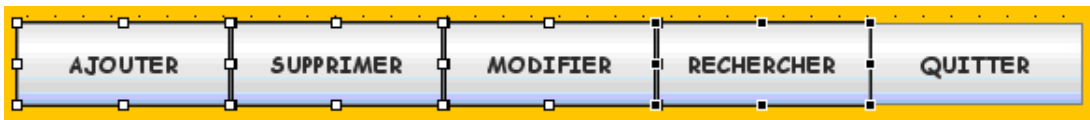
- Créez maintenant une copie de la classe **FenTableClient** et nommez-la comme indiqué précédemment **FenTableN2Client**. Veillez à ce que cette nouvelle fenêtre soit rangée dans le paquetage **dialogue**.
- Changez la couleur du fond de la fenêtre **FenTableN2Client**. Par exemple :

```

jContentPane.setBackground(Color.orange);

```

- Ouvrez cette classe avec Visual Editor et supprimez tous les boutons sauf le bouton **QUITTER**.



- Modifiez son constructeur.

```

public FenTableClientN2(DefaultTableModel leModeleCommun) {
    super();
    leModeleClients = leModeleCommun;
    initialize();
}

```



Cette fenêtre recevra le modèle de la fenêtre **FenTableClient**. Elles sont ainsi concernées par le même modèle et toute modification concernant ce dernier est répercutée sur ces deux vues.

- Modifiez sa méthode **remplirTable()**.

```

private JTable remplirTable(){
    laTable_Table = new JTable(leModeleClients);
    laTable_Table;
}

```

- Supprimez pour finir sa méthode **tableChanged(...)**.
- Ouvrez maintenant la méthode **FenTableClient**. Modifiez la méthode **getBtn_Ajouter()**.

```

private JButton getBtn_Ajouter() {
    if (btn_Ajouter == null) {
        ...
        public void mouseClicked(java.awt.event.MouseEvent e) {
            ...
            laFiche_client.setVisible(true);

            // création de la 2ème fenêtre avec le même modèle
            FenTableClientN2 laDeuxiemeFenTable = new FenTableClientN2
(leModeleClients);
            laDeuxiemeFenTable.setVisible(true);
        }
    }
}

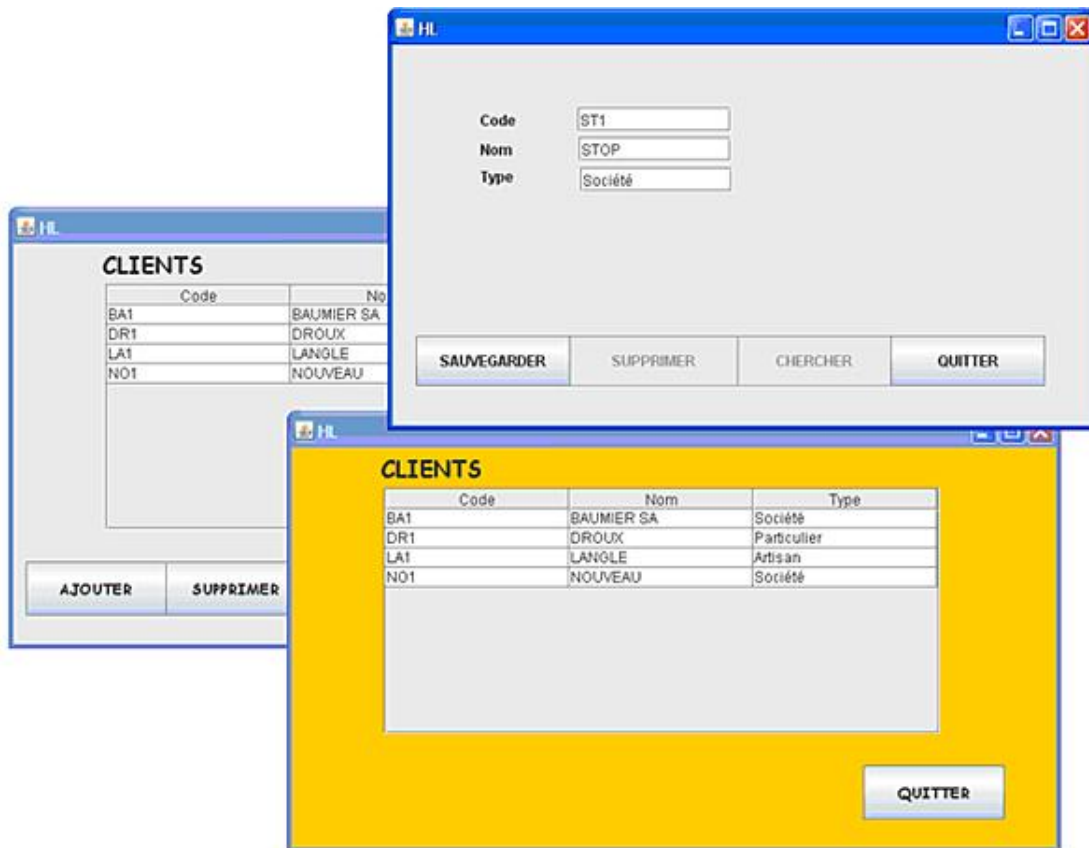
```

```

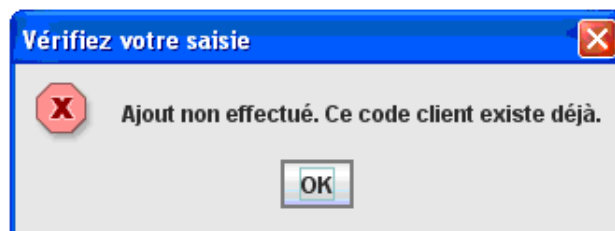
    });
}
return btn_Ajouter;
}

```

- Testez l'application en ajoutant des clients.



La tentative d'ajout d'un nouveau client avec un code existant déjà dans la base de données est refusée et déclenche l'affichage ci-après :



L'ajout d'une nouvelle ligne dans le modèle est cependant effectué. Pour y remédier, il faut retourner le résultat de la tentative d'ajout d'un nouvel enregistrement dans la base de données. Trois classes sont concernées :

- la classe **Client** ;
 - la classe **GestionDemandes** ;
 - la classe **FenFicheClient**.
- Procédez aux modifications suivantes :
- dans la classe **Client** :

```
// Ajout d'un nouveau client dans la BD
public boolean creerCRUD_Client(Client leClient){
    boolean bCreation = true;

    String requete = null;
    try {
        requete = "INSERT INTO client(Code, Nom, Type)" +
            " VALUES('"+super.getCode()+"', '"+super.getNom()+"',
            '"+type+"')";
        leModele.fireTableDataChanged();
        Statement state = laConnexion.createStatement();
        state.executeUpdate(requete);
        state.close();
    }
    catch (SQLException e){
        bCreation = false;
        JOptionPane.showMessageDialog(null, "Ajout non effectué."
            + " Ce code client existe déjà.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
    }
    return bCreation;
}
```

- dans la classe **GestionDemandes** :

```
// Demande d'ajout d'un nouveau client dans la BD
public boolean demandeEnregistrerClient(String vCode, String vNom,
String vType){
    boolean bCreation = false;
    Client leClient = new Client(vCode, vNom, vType);
    bCreation = leClient.creerCRUD_Client(leClient);return bCreation;
}
```

- dans la classe **FenFicheClient** :

```
private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        ...
        btn_Enregistrer.addActionListener(new java.awt.event.Action
        Listener() {
            public void actionPerformed(java.awt.event.ActionEvent e){
                ...
                if (btn_Enregistrer.getText()=="SAUVEGARDER"){
                    if(!vCode.equals("")){
                        // demande d'ajout d'un client dans la BDboolean bCreation ;
                        bCreation =
                        leClientGestionClientBD.demandeEnregistrerClient(vCode, vNom,
                            vType);if (bCreation == true){
                            jTxTCode.setText("");
                            ...
                            jTxTCode.requestFocusInWindow();
                            Vector<String> ligne = new Vector<String>();
                            leModeleTransmisParFenTable.addRow(ligne);
                            ...
                            leModeleTransmisParFenTable.setValueAt(vType, nb
                            nbLignes, 2);
                        }
                    }
                }
            }
            else {
                JOptionPane.showMessageDialog(null, "La saisie du"+
                    + "code client"
                    + " est obligatoire",
                    "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}
```



```

    }
    }
    // sinon nous sommes en mode modification
    else {
        leClientGestionClientBD.demandeModifierClient(vCode,
vNom, vType);
    }
    }
    });
}
return btn_Enregistrer;
}

```

2. Suppression d'un enregistrement

Nous suivons la même démarche que pour l'ajout. Les deux méthodes ci-après retournent désormais un booléen.

- Ouvrez la classe **Client** et modifiez la méthode **supprimerCRUD_Clients(Client leClient)**. Celle-ci retourne désormais un booléen.

```

public boolean supprimerCRUD_Client(Client leClient){
    boolean bSuppression = true;
    ...
    try {
        ...
        if (nbEnregSup == 0){
            ...
        }
        else {
            ...
        }
    }
    catch (SQLException e){
        ...
    }return bSuppression;
}

```

- Ouvrez la classe **GestionDemandes** et modifiez la méthode **demandeSupprimerClient(String vCode)**. Celle-ci retourne désormais un booléen.

```

public boolean demandeSupprimerClient(String vCode){
    boolean bSuppression = false;
    Client leClient = new Client(vCode);
    bSuppression = leClient.supprimerCRUD_Client(leClient);return bSuppression;
}

```

- Ouvrez maintenant la classe **FenFicheClient** et modifiez la méthode **JButton getBtn_ Supprimer()**.

```

private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        ...
        btn_Supprimer.addActionListener
(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String vCode = jTxTCode.getText();
                if(!vCode.equals("")){boolean bSuppression ;
                    ...
                    if (choix==0){
                        ...
                        if(bSuppression==true)
                            int i = 1;

```

```

        int ligne = 0;
        int colonne = 0;
        boolean bTrouve = false;
        String leCode;
        while(i <= leModeleTransmisParFenTable.getRowCount() &&
bTrouve==false){
            leCode=(String)leModeleTransmisParFenTable.getValueAt
(ligne, colonne);
            if(leCode.equals(vCode)){
                bTrouve = true;
                leModeleTransmisParFenTable.removeRow(i-1);
            }
            i++;
            ligne++;
        }
    }
    else {
        ...
    }
}
});
return btn_Supprimer;
}

```

 La méthode **getValueAt(int rowIndex, int columnIndex)** de l'interface **TableModel** retourne un objet. Il faut donc transtyper en chaîne la valeur retournée.

Ne pas oublier que l'index des lignes du modèle commence à zéro.

- Pour finir, dans le code de la classe **FenTableClient**, recherchez la méthode **getBtn_Supprimer()** puis désactivez ou supprimez la ligne qui referme la fenêtre. Insérez aussi une ligne de code pour s'assurer que le bouton de suppression demeure actif.

```

private JButton getBtn_Supprimer() {
    ...
    if(NumLigne == -1){
        // dispose();  à désactiver ou à supprimer
        // on active le bouton SUPPRIMER de la fiche client
        laFiche_client.setBtn_SupprimerActif();
    }
    ...
}

```

- Testez la suppression.

Nous allons poursuivre les tests avec plusieurs fenêtres.

- Ajoutez le code suivant à la méthode **getBtn_Supprimer()**.

```

private JButton getBtn_Supprimer() {
    ...
    if(NumLigne == -1){
        // dispose();  à désactiver ou à supprimer
        // on active le bouton SUPPRIMER de la fiche client
        laFiche_client.setBtn_SupprimerActif();
        FenTableClientN2 laDeuxiemeFenTable =
new FenTableClientN2(leModeleClients);
        laDeuxiemeFenTable.setVisible(true);
    }
    ...
}

```

- Testez la suppression avec plusieurs fenêtres ouvertes.

3. Modification d'un enregistrement

La démarche reste la même. Nous avons à intervenir sur les mêmes classes :

- **Client** ;
- **GestionDemandes** ;
- **FenFicheClient**.

■ Procédez aux modifications suivantes :

- dans la classe **Client** :

```
public boolean modifierCRUD_Client(Client leClient){
    boolean bModification = true;
    String requete = null;
    try {
        ...
    }
    catch (SQLException e){
        bModification = false;
        ...
    }
    return bModification;
}
```

- dans la classe **GestionDemandes** :

```
public boolean demandeModifieurClient(String vCode, String vNom,
String vType){
    boolean bModification = false;
    Client leClient = new Client(vCode, vNom, vType);
    bModification = leClient.modifierCRUD_Client(leClient);return bModification;
}
```

- dans la classe **FenFicheClient** :

```
private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        ...
        btn_Enregistrer.addActionListener
new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent e){
        ...


        // sinon nous sommes en mode modification
        else {boolean bModification ;
            bModification =
leClientGestionClientBD.demandeModifieurClient(vCode, vNom, vType);
            if(bModification == true){
                int i = 1;
                int ligne = 0;
                int colonne = 0;
                boolean bTrouve = false;
                String leCode;
                while(i <= leModeleTransmisParFenTable.getRowCount() &&
bTrouve==false){
                    leCode=
                    (String)leModeleTransmisParFenTable.getValueAt(ligne, colonne);
```

```

        if (leCode.equals(vCode)) {
            bTrouve = true;
            leModeleTransmisParFenTable.setValueAt(vCode, ligne, 0);
            leModeleTransmisParFenTable.setValueAt(vNom, ligne, 1);
            leModeleTransmisParFenTable.setValueAt(vType, ligne, 2);
        }
        i++;
        ligne++;
    }

    ...
    return btn_Enregistrer;
}

```

 Rappelons que le bouton **btn_Enregistrer** sert à la fois pour les créations et les enregistrements. Comme pour les suppressions, les modifications nécessitent un parcours du modèle.

4. Recherche d'enregistrements

- Effectuez une recherche.

La fenêtre **FenResultatRecherche** ne présente plus aucune donnée quel que soit le résultat de la recherche. Ceci est dû au fait que la méthode **chercherCRUD_Clients(Client lesClients)** de la classe **Client** a été modifiée pour remplir un modèle, instance de la classe **DefaultTableModel**. Le composant **JTable** doit donc recevoir ce modèle en paramètre.

- Ouvrez la classe **FenResultatRecherche** et modifiez la méthode **remplirTable()** :

```

private JTable remplirTable(){
    Client lesClients = new Client(leCode);
    lesClients.chercherCRUD_Clients(lesClients);
    DefaultTableModel leModeleClients = lesClients.getLeModeleClients();
    laTable_Table = new JTable(leModeleClients);
    return laTable_Table;
}

```

- Testez la recherche.

Code complet de la classe **Client** version MVC :

```

package entite;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.Vector;

import controle.*;

import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

public class Client extends Personne{
    // Propriétés
    // *****
    private DefaultTableModel leModele = new DefaultTableModel();
    private String type;
    private static Connection laConnexion = ControleConnexion.getConnexion();
    private Vector<Vector> tabLignes = new Vector<Vector>();
    private Vector<String> nomColonnes = new Vector<String>();
}

```

```

// Constructeurs
// *****
// 1er Constructeur
public Client(String vCode, String vNom, String vType){
    super(vCode, vNom);
    type = vType;
}
// 2ème Constructeur
public Client(String vCode){
    super(vCode);
}

// Méthodes
// *****
public Vector<Vector> getTabLignes(){
    return tabLignes;
}
public Vector<String> getNomColonnes(){
    return nomColonnes;
}
public String getType(){
    return type;
}
public void setType(String vTitre){
    type = vTitre;
}
// accesseur pour retourner le modèle
public DefaultTableModel getLeModeleClients(){
    return leModele;
}

// Ajout d'un nouveau client dans la BD
// -----
public boolean creerCRUD_Client(Client leClient){
    boolean bCreation = true;
    String requete = null;
    try {
        requete = "INSERT INTO client(Code, Nom, Type)" +
            " VALUES('"+super.getCode()+"', '"+super.getNom()+"', '"+type+"')";
        Statement state = laConnexion.createStatement();
        state.executeUpdate(requete);
        state.close();
    }
    catch (SQLException e){
        bCreation = false;
        JOptionPane.showMessageDialog(null, "Ajout non effectué."
            + " Ce code client existe déjà.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        System.out.println("Insertion non effectuée");
    }
    return bCreation;
}

// Modification d'un client dans la BD
// -----
public boolean modifierCRUD_Client(Client leClient){
    boolean bModification = true;
    String requete = null;
    try {
        requete = "UPDATE client SET Nom = '" + leClient.getNom()+"' "
            + "Type = '" + leClient.getType()+"' "
            + " WHERE Code = '" + leClient.getCode()+"'";
        Statement state = laConnexion.createStatement();
        state.executeUpdate(requete);
        state.close();
        JOptionPane.showMessageDialog(null, "Modification réalisée",
            "MODIFICATION", JOptionPane.INFORMATION_MESSAGE);
    }
    catch (SQLException e){
        bModification = false;
        JOptionPane.showMessageDialog(null, "Modification non effectuée",
            "MODIFICATION", JOptionPane.ERROR_MESSAGE);
    }
    return bModification;
}

```

```

    }
    catch (SQLException e){
        bModification = false;
        JOptionPane.showMessageDialog(null, "Modification non effectuée."
            + " Ce code client n'existe pas.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        System.out.println("Modification non effectuée");
    }
    return bModification;
}

// Suppression d'un client dans la BD
// -----
public boolean supprimerCRUD_Client(Client leClient){
    boolean bSuppression = true;
    String leCode = leClient.getCode();
    String requete = null;
    try {
        // NE PAS OUBLIER L'ESPACE DEVANT WHERE APRES UN + !!!
        requete = "DELETE FROM client" +
        " WHERE Code = '" + leCode + "'";
        Statement state = laConnexion.createStatement();
        int nbEnregSup = state.executeUpdate(requete);
        if (nbEnregSup == 0){
            JOptionPane.showMessageDialog(null,
"Aucune suppression effectuée."
            + " Ce code client n'existe pas.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        }
        else {
            state.close();
            JOptionPane.showMessageDialog(null, "Suppression du client code ["
            + leCode + "] effectuée");
        }
    }
    catch (SQLException e){
        bSuppression = false;
        JOptionPane.showMessageDialog(null, "Problème lors de la" +
"suppression.",
            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
    }
    return bSuppression;
}

// Chercher un ou plusieurs clients dans la BD
// -----
public int chercherCRUD_Clients(Client lesClients){
    String leCode = lesClients.getCode();
    int i, nbClients=0;
    String requete = null;
    try {
        leCode = leCode + "%";
        requete = "SELECT * FROM client WHERE Code LIKE '" + leCode + "'"
        + " ORDER BY code, nom";
        Statement state = laConnexion.createStatement();
        ResultSet jeuEnregistrements = state.executeQuery(requete);
        ResultSetMetaData infojeuEnregistrements =
jeuEnregistrements.getMetaData();
        for( i=1; i <= infojeuEnregistrements.getColumnCount(); i++){
            String nomColonneModel =
infojeuEnregistrements.getColumnLabel(i);
            leModele.addColumn(nomColonneModel);
        }
        while(jeuEnregistrements.next()) {
            Vector<String> ligne = new Vector<String>();
            for(i=1; i <= infojeuEnregistrements.getColumnCount();
i++) {

```

```

        String chaine_champ = jeuEnregistrements.getString(i);
        ligne.add(chaine_champ);
    }
    nbClients = nbClients + 1;
    leModele.addRow(ligne);
}
jeuEnregistrements.close();
state.close();
}

catch (SQLException e){
    JOptionPane.showMessageDialog(null, "Problème lors de la" +
"recherche.", "",
        JOptionPane.ERROR_MESSAGE);
}
return nbClients;
}
}

```

Code complet de la classe **GestionDemandes** version MVC :

```

package controle;

import entite.Client;

public class GestionDemandes {
    // CONSTRUCTEUR
    // *****
    public GestionDemandes(){
    }

    // METHODES
    // *****
    // demande d'ajout d'un nouveau client dans la BD
    // -----
    public boolean demandeEnregistrerClient(String vCode, String vNom,
String vType){
        boolean bCreation = false;
        Client leClient = new Client(vCode, vNom, vType);
        bCreation = leClient.creerCRUD_Client(leClient);
        return bCreation;
    }

    // demande de modification d'un client dans la BD
    // -----
    public boolean demandeModifierClient(String vCode, String vNom,
String vType){
        boolean bModification = false;
        Client leClient = new Client(vCode, vNom, vType);
        bModification = leClient.modifierCRUD_Client(leClient);
        return bModification;
    }

    // demande de suppression d'un client dans la BD
    // -----
    public boolean demandeSupprimerClient(String vCode){
        boolean bSuppression = false;
        Client leClient = new Client(vCode);
        bSuppression = leClient.supprimerCRUD_Client(leClient);
        return bSuppression;
    }

    // demande de recherche d'un ou plusieurs clients dans la BD
    // -----
    public int demandeChercherClient(String vCode){
        Client leClient = new Client(vCode);
        return leClient.chercherCRUD_Clients(leClient);
    }
}

```

```
}
```

Code complet de la classe **FenFicheClient** version MVC :

```
package dialogue;

import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;
import java.awt.Rectangle;
import java.util.Vector;

import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.table.DefaultTableModel;

import controle.GestionDemandes;

public class FenFicheClient extends JFrame {
    // Propriétés non graphiques
    // *****
    private DefaultTableModel leModeleTransmisParFenTable;
    private GestionDemandes leClientGestionClientBD = new GestionDemandes();

    // Propriétés graphiques
    // *****
    private static final long serialVersionUID = 1L;
    private JPanel jContentPane = null;
    private JTextField jTxTCode = null;
    private JTextField jTxTNom = null;
    private JTextField jTxTType = null;
    private JLabel jLabelCode = null;
    private JLabel jLabelNom = null;
    private JLabel jLabelType = null;
    private JButton btn_Enregistrer = null;
    private JButton btn_Quitter = null;
    private JButton btn_Supprimer = null;
    private JButton btn_Rechercher = null;

    // Constructeur
    // *****
    public FenFicheClient(DefaultTableModel leModele) {
        super();
        initialize();
        leModeleTransmisParFenTable = leModele;
    }

    // Méthodes
    // *****
    private void initialize() {
        this.setSize(579, 344);
        this.setContentPane(getJContentPane());
        this.setTitle("HL");
        // pour centrer la fenêtre sur l'écran
        this.setLocationRelativeTo(null);
        this.setVisible(false);
    }
    private JPanel getJContentPane() {
        if (jContentPane == null) {
            jLabelType = new JLabel();
            jLabelType.setBounds(new Rectangle(72, 99, 76, 16));
            jLabelType.setText("Type");
            jLabelNom = new JLabel();
            jLabelNom.setBounds(new Rectangle(72, 77, 78, 16));
            jLabelNom.setText("Nom");
        }
    }
}
```



```

        jLabelCode = new JLabel();
        jLabelCode.setBounds(new Rectangle(72, 53, 77, 16));
        jLabelCode.setText("Code");
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
        jContentPane.add(getJTxTCode(), null);
        jContentPane.add(getJTxTNom(), null);
        jContentPane.add(getJTxTType(), null);
        jContentPane.add(jLabelCode, null);
        jContentPane.add(jLabelNom, null);
        jContentPane.add(jLabelType, null);
        jContentPane.add(getBtn_Enregistrer(), null);
        jContentPane.add(getBtn_Quitter(), null);
        jContentPane.add(getBtn_Supprimer(), null);
        jContentPane.add(getBtn_Rechercher(), null);
    }
    return jContentPane;
}

public JTextField getJTxTCode() {
    if (jTxTCode == null) {
        jTxTCode = new JTextField();
        jTxTCode.setBounds(new Rectangle(151, 51, 125, 20));
    }
    return jTxTCode;
}

public JTextField getJTxTNom() {
    if (jTxTNom == null) {
        jTxTNom = new JTextField();
        jTxTNom.setBounds(new Rectangle(152, 75, 124, 20));
    }
    return jTxTNom;
}

public JTextField getJTxTType() {
    if (jTxTType == null) {
        jTxTType = new JTextField();
        jTxTType.setBounds(new Rectangle(153, 100, 124, 20));
    }
    return jTxTType;
}

// ajout des mutateurs pour que les champs puissent être modifiés
// depuis la fenêtre FenTableMenu
public void setJTxTCode(String vCode) {
    jTxTCode.setText(vCode);
}

public void setJTxTNom(String vNom) {
    jTxTNom.setText(vNom);
}

public void setJTxTType(String vType) {
    jTxTType.setText(vType);
}

public void setBtn_EnregistrerLibelle(String vLibelle) {
    btn_Enregistrer.setText(vLibelle);
}

// SAUVEGARDE & MODIFICATION
// -----
private JButton getBtn_Enregistrer() {
    if (btn_Enregistrer == null) {
        btn_Enregistrer = new JButton();
        btn_Enregistrer.setBounds(new Rectangle(19, 237, 126, 41));
        btn_Enregistrer.setText("SAUVEGARDER");
        btn_Enregistrer.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String vCode = jTxTCode.getText();
                String vNom = jTxTNom.getText();
            }
        });
    }
    return btn_Enregistrer;
}

```

```

String vType = jTxTType.getText();
if (btn_Enregistrer.getText()=="SAUVEGARDER"){
    if (!vCode.equals("")){
        boolean bCreation ;
        bCreation = leClientGestionClientBD.demandeEnregistrerClient
(vCode,vNom, vType);
        if (bCreation == true){
            jTxTCode.setText("");
            jTxTNom.setText("");
            jTxTType.setText("");
            jTxTCode.requestFocusInWindow();

            Vector<String> ligne = new Vector<String>();
            leModeleTransmisParFenTable.addRow(ligne);
            int nbLignes = leModeleTransmisParFenTable.getRowCount();
            nbLignes = nbLignes -1;
            leModeleTransmisParFenTable.setValueAt(vCode, nbLignes, 0);
            leModeleTransmisParFenTable.setValueAt(vNom, nbLignes, 1);
            leModeleTransmisParFenTable.setValueAt(vType, nbLignes, 2);
        }
    }
    else {
        JOptionPane.showMessageDialog(null, "La saisie du code client"
+ " est obligatoire",
"Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
    }
}
else {
    boolean bModification ;
    bModification = leClientGestionClientBD.demandeModifierClient
(vCode, vNom, vType);
    if(bModification == true){
        int i = 1;
        int ligne = 0;
        int colonne = 0;
        boolean bTrouve = false;
        String leCode;
        while(i <= leModeleTransmisParFenTable.getRowCount())&&
bTrouve==false){
            leCode=(String) leModeleTransmisParFenTable.getValueAt(ligne,
colonne);
            System.out.println("ligne : "+ ligne + " --> code : " +
leCode);
            if(leCode.equals(vCode)){
                bTrouve = true;
                System.out.println("trouvé --> ligne à modifier : " +
(i-1));
                leModeleTransmisParFenTable.setValueAt(vCode, ligne, 0);
                leModeleTransmisParFenTable.setValueAt(vNom, ligne, 1);
                leModeleTransmisParFenTable.setValueAt(vType, ligne, 2);
            }
            i++;
            ligne++;
        }
    }
}
});
}
return btn_Enregistrer;
}

private JButton getBtn_Supprimer() {
    if (btn_Supprimer == null) {
        btn_Supprimer = new JButton();
        btn_Supprimer.setBounds(new Rectangle(144, 237, 138, 41));
        btn_Supprimer.setText("SUPPRIMER");
    }
}

```

```

        btn_Supprimer.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String vCode = jTxTCode.getText();
                if(!vCode.equals("")){
                    boolean bSuppression ;
                    int choix = JOptionPane.showConfirmDialog(null,
                        "Voulez-vous supprimer la fiche du client code : " + vCode + " ?",
                        "Suppression",
                        JOptionPane.YES_NO_OPTION);
                    if (choix==0){
                        bSuppression =
leClientGestionClientBD.demandeSupprimerClient(vCode);
                        if(bSuppression==true)
                            jTxTCode.setText("");
                            jTxTCode.requestFocusInWindow();
                            int i = 1;
                            int ligne = 0;
                            int colonne = 0;
                            boolean bTrouve = false;
                            String leCode;
                            while(i <= leModeleTransmisParFenTable.getRowCount() &&
bTrouve==false){
                                leCode=
(String)leModeleTransmisParFenTable.getValueAt(ligne, colonne);
                                if(leCode.equals(vCode)){
                                    bTrouve = true;
                                    leModeleTransmisParFenTable.removeRow(i-1);
                                }
                                i++;
                                ligne++;
                            }
                        }
                    }
                    else {
                        JOptionPane.showMessageDialog(null, "La saisie du code client"
                            + " est obligatoire",
                            "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
                    }
                }
            }
        });
    }
    return btn_Supprimer;
}

/*
 * RECHERCHE
 */
private JButton getBtn_Rechercher() {
    if (btn_Rechercher == null) {
        btn_Rechercher = new JButton();
        btn_Rechercher.setBounds(new Rectangle(281, 237, 126, 41));
        btn_Rechercher.setText("CHERCHER");
        btn_Rechercher.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                String vCode = jTxTCode.getText();
                String vNom = jTxTNom.getText();
                String vType = jTxTType.getText();
                if(!vCode.equals("") || !vNom.equals("") || !vType.equals("")){
                    int nbEnreg =
leClientGestionClientBD.demandeChercherClient(vCode);
                    if(nbEnreg > 0){
                        switch(nbEnreg)
                        {
                            case 0:
                                JOptionPane.showMessageDialog(null, "Aucun client trouvé."
                                    + " Ce code client n'existe pas.",

```

```

        "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
        break;
    case 1:
        JOptionPane.showMessageDialog(null, nbEnreg +
" client trouvé",
        "RESULTAT", JOptionPane.INFORMATION_MESSAGE);
        break;
    default:
        JOptionPane.showMessageDialog(null, nbEnreg +
" clients trouvés.",
        "RESULTAT", JOptionPane.INFORMATION_MESSAGE);
    }
    btn_Enregistrer.setEnabled(true);
    jTxTCode.setText(vCode);
    new FenResultatRecherche(vCode);
}
if(nbEnreg == 0){
    JOptionPane.showMessageDialog(null, "Aucun client trouvé."
    + " Ce code client n'existe pas.",
    "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
}
}
else {
    JOptionPane.showMessageDialog(null, "Saisissez au moins"
    + " un critère de recherche",
    "Vérifiez votre saisie", JOptionPane.ERROR_MESSAGE);
}
}
});
}
return btn_Rechercher;
}

// GESTION DE L'ACCESSIBILITE DES BOUTONS EN FONCTION DES CHOIX
// -----
public JButton setBtn_EnregistrerActif() {
    btn_Enregistrer.setEnabled(true);
    return btn_Enregistrer;
}
public JButton setBtn_EnregistrerNonActif() {
    btn_Enregistrer.setEnabled(false);
    return btn_Enregistrer;
}

public JButton setBtn_SupprimerActif() {
    btn_Supprimer.setEnabled(true);
    return btn_Supprimer;
}
public JButton setBtn_SupprimerNonActif() {
    btn_Supprimer.setEnabled(false);
    return btn_Supprimer;
}

public JButton setBtn_RechercherActif() {
    btn_Rechercher.setEnabled(true);
    return btn_Rechercher;
}
public JButton setBtn_RechercherNonActif() {
    btn_Rechercher.setEnabled(false);
    return btn_Rechercher;
}

public void changeTitre(String leTitre){
    btn_Enregistrer.setText(leTitre);
}

private JButton getBtn_Quitter() {
    if (btn_Quitter == null) {

```

```
btn_Quitter = new JButton();
btn_Quitter.setBounds(new Rectangle(406, 237, 126, 41));
btn_Quitter.setText("QUITTER");
btn_Quitter.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent e) {
        FenFicheClient.this.dispose();
    }
});
return btn_Quitter;
}
```